Applied Computer Systems
ISSN 2255-8691 (online)
ISSN 2255-8683 (print)
December 2023, vol. 28, no. 2, pp. 221–231
https://doi.org/10.2478/acss-2023-0022
https://content.sciendo.com

Source Code Features and their Dependencies: An Aggregative Statistical Analysis on Open-Source Java Software Systems

Farshad Ghassemi Toosi[*]
Department of Computer Science, Munster Technological University, Cork, Ireland

Abstract **– Source code constitutes the static and human-readable component of a software system. It comprises an array of artifacts and features that collectively execute a specific set of tasks. Coding behaviours and patterns are formulated through the orchestrated utilization of distinct features in a specified sequence, fostering inter-dependencies among these features. This study seeks to explore into the presence of specific coding behaviours and patterns within Java, which could potentially unveil the extent to which developers endeavour to leverage the facilities and services that exist in the programming language aggregatively. In pursuit of investigating behaviours and patterns, 436 open-source Java projects are selected, each having more than 150 Java files (Classes and Interfaces), in a semi-randomized manner. For every project, 39 features have been chosen, and the frequency of each individual feature has been independently assessed. By employing linear regression, the interrelationships among all features across the complete array of projects are scrutinized. This analysis intends to uncover the manifestation of distinct coding behaviours and patterns. Based on the selected features, preliminary findings suggest a notable collective incorporation of diverse coding behaviours among programmers, encompassing Encapsulation and Polymorphism. The findings also point to a distinct preference for using a specific commenting mechanism, JavaDoc, and the potential existence of Code-Clone and dead code. Overall, the results indicate a clear tendency among programmers to strongly adhere to the fundamental principles of Object-Oriented programming. However, certain less obvious attributes of object-oriented languages appear to receive relatively less attention from programmers.**

Keywords **– Object oriented principle, source code analysis, source code feature extraction.**

I. Introduction

The source code of a software system encompasses diverse artifacts, components, and features that empower programmers to configure code structures, patterns, behaviours, and styles.

These artifacts and characteristics have unique qualities and purposes, but they can also show connections in certain situations as indicated by De Moura et al. [1]. For example, in object-oriented programming languages like Java [2], a class that contains many private fields would require a corresponding number of methods to manage those fields (Encapsulation). Java offers essential tools for creating software systems that are maintainable and can be reused, provided that best practices are carefully followed.

Background

Analysing the source code of a software system, either individually or collectively, can reveal various details that can benefit developers, both for existing systems and newly created ones. Miltiadis et al. [3] conducted an experiment involving an extensive codebase, comprising approximately 352 million lines of Java, to analyse the complexity of a code module and its topical centrality within a software project. Their method assists in distinguishing reusable utility classes from those that constitute a program's core logic, solely based on general information-theoretic criteria.

O'Hare et al. [4] developed an analyser called CASE (Computer-Aided Software Engineering), which essentially functions as a reverse engineering system. It aids in discovering the structure of a software system from its source code. This analyser has the potential to assist in transforming legacy code into abstractions within a structured analysis methodology.

Source code analysis also has applications in cybersecurity and vulnerability detection [5], [6]. In a study by Arvinder et al. [7], code analysers for three languages, namely C, C++, and Java, have been compared and discussed.

In another study, Johnatan et al. [8] conducted an experiment applying source code analysis to assess developers' skills. They evaluated two models designed to support the realization of programming skills. Their research involved a survey with 110 developers from GitHub, conducted to assess the applicability of these two models for computing developers' programming skills based on the metrics 'Changed Files' and 'Changed Lines of Code'.

Source code analysis has also application in specific-domain languages. Ivan et al. [9], in their work, discuss the concerns related to evaluate the quality of such languages and they propose a model-driven interoperability strategy that bridges the gap between the grammar formats of source code quality parsers and domain-specific text languages.

Static source code analysis can help the errors and issues to be raised from multi-threading programming that usually is

[*] Corresponding author's e-mail: farshad.toosi@mtu.ie

clear when the application is deployed. Damian et al. [10] discuss some of these errors such as race condition, deadlock, atomicity violation and order violation. They also present a model that can detect such errors using static source code analysis techniques.

In the course of this study, a protocol for source code analysis has been devised, aimed at probing the extent of interdependence among a specific set of chosen code "features". The principal objective of this work is to explore contemporary coding practices within Java from a distinctive point of view, with the intention of deriving an overview of prevalent behaviours on an aggregate level. In essence, the overall aim is to discover whether feature statistics can offer insights into coding practices or programming concepts (e.g., Object Oriented Concepts).

II. Design and Methods

In this study, a total of 436 open-source Java projects were collected from GitHub using the GitHubBuilder [11] API. The GitHubBuilder API enables the search of open-source GitHub projects based on specified queries. The selected queries for this study included terms such as market, customer, algorithm, and network. By employing these queries, projects containing any of these keywords are identified. The GitHubBuilder API can specifically target projects written in certain programming languages, with Java being the language of focus in this study.

Each detected project underwent an initial examination, and if it contained more than 150 Java files (Class, Abstract Class, and Interface), it was added to a pool of Java projects for further analysis. For each project, 39 features were considered, namely: Method-Call, All-Methods, Accessed-Variable, Declared-Variable, All-Comment, Line-Comment, Block-Comment, Doc-Comment (also known as JavaDoc), Inner-Class (including inner and/or nested classes), LOC (Lines Of Code), Class-Call (object creation), Inheritance-Direct, Inheritance-All (covering directed and indirect inheritance, such as a SuperClass of a SuperClass), Exception (Try blocks), Catch (Catch blocks), Concrete (concrete class), Abstract (Abstract class), Interfaces, All-Class-Type (covering concrete, abstract, and interface), FinalMethod, Overloaded-Method, Overridden-Method, Default-Method, Private-Method, Protected-Method, Public-Method, Static-Method, Super-Method (including abstract methods and those that are overridden), Variable-Final, Variable-Primitive, VariableDefault, Variable-Private, Variable-Protected, Variable-Public, Variable-Static, Ave-Class-LOC, Ave-Method-LOC, Dead-Method (a type of dead code), and Package-Count. These features were chosen for their popularity in coding practices and their inclusion in object-oriented mechanisms.

The frequency of each feature is counted across the entire project. For example, in a given project, the count of Inheritance-Direct is calculated for each class and then summed across the entire project for all classes and interfaces.

Given that there are a total of 436 projects, each of the aforementioned features can be treated as a vector with a size of 436. To explore the relationships between any pair of features, linear regression is applied. For a given pair of features F_1 and F_2, a 2D line can be drawn using these two features as x and y, as shown in Fig. 2. The other image in Fig. 1 depicts a straight line, indicating complete linear dependency between F_1 and F_2; whereas the right image portrays a broken line, suggesting little to no linear dependency between F_1 and F_2.

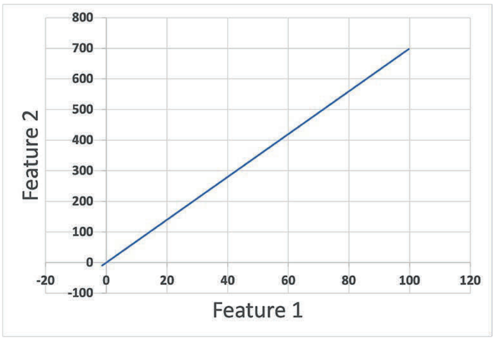


Fig. 1. An example of a straight line with a 100 % correlation between two features: Feature 1 and Feature 2.

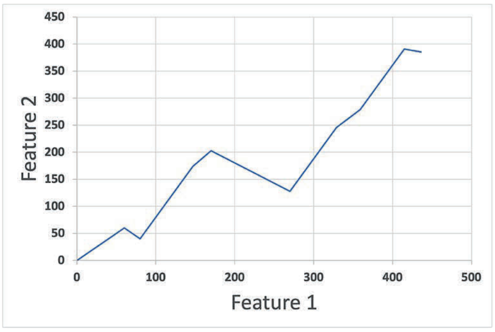


Fig. 2. An example of a broken line: The X-axis represents Feature 1, and the Y-axis represents Feature 2.

The line's degree of straightness is quantified using linear regression techniques. The Pearson Correlation Coefficient (PCC) [12] is a real value denoted as r, which lies within the range of $[-1,1]$. A value of -1 indicates the maximum negative linear correlation, while a value of 1 signifies the maximum positive linear correlation between two variables. Values close to 0 indicate a weaker linear correlation, and a value of 0 represents no correlation.

III. Relationships

The objective of this study is to investigate interdependencies among features. For instance, from an object-oriented perspective, it is expected that a project with a higher number of inheritance will have a higher number of overridden methods. Therefore, there is an expected dependency between the number of inheritance (F_1) and the number of overridden methods (F_2). Through this analysis, these dependencies can be explored, and more relationships may be uncovered.

To conduct this analysis, the Pearson Correlation Coefficient (PCC) is employed. Schober et al. [13] discussed the details of the Correlation Coefficient in their work and presented Table I as a Conventional Approach to interpreting a Correlation Coefficient. Consequently, the information in Table I is used as a reference for interpretation. As this study aims to identify dependent features, features exhibiting a Strong Correlation

(0.70–0.89) or Very Strong Correlation (0.90–1.00) are discussed in detail.

TABLE I

INTERPRETATION OF PCC

Absolute Value of PCC	Interpretation
0.00–0.09	Negligible correlation
0.10–0.39	Weak correlation
0.40–0.69	Moderate correlation
0.70–0.89	Strong correlation
0.90–1.00	Very strong correlation

A. Discussion

In this section, each feature is individually examined, and the most dependent features on it are listed and discussed. Table II presents the list of features, with each feature assigned an index.

Figures 19 to 28 illustrate individual feature dependencies. As previously mentioned, these figures only show strong dependencies. The complete list of dependencies for all features can be observed in Fig. 29, and the reference indexing is provided in Table II.

TABLE II

LIST OF ALL FEATURES AND THEIR INDEX

1	Method-Call	14	Exception	27	Static-Method
2	All-Methods	15	Catch	28	Super-Method
3	Accessed-Variable	16	Concrete	29	Variable-Final
4	Declared-Variable	17	Abstract	30	Variable-Primitive
5	All-Comment	18	Interfaces	31	Variable-Default
6	Line-Comment	19	All-Class-Type	32	Variable-Private
7	Block-Comment	20	Final-Method	33	Variable-Protected
8	Doc-Comment	21	Overloaded-Method	34	Variable-Public
9	Inner-Class	22	Overridden-Method	35	Variable-Static
10	LOC	23	Default-Method	36	Ave-Class-LOC
11	Class-Call	24	Private-Method	37	Ave-Method-LOC
12	Inheritance-Direct	25	Protected-Method	38	Dead-Method
13	Inheritance-All	26	Public-Method	39	Package-Count

- *LOC*: Lines of Code simply represents the number of lines of code present in the software system. LOC is employed as a metric to measure the size of the source code or the software system itself [14], [15]. Figure 18 illustrates the strongly dependent features of LOC, with All-Methods and All-Variables (Declared Variables) being the two most pronounced dependencies. Therefore, All-Methods and All-Variables are going to be considered as project-size metrics for future features.
- *Encapsulation*: Encapsulation is one of the fundamental Object-Oriented Principles. It serves as a mechanism to prevent direct access to class data from other classes. Instead, class data may be accessed through methods, allowing for any desirable conditions to be applied when accessing the data. Typically, encapsulation is realized by defining class fields as private variables and creating methods to access those private variables. Statistically, under an ideal scenario, the hypothesis is formulated as follows: As the size of the software increases, instances of encapsulation also increase. To validate this hypothesis, the presence of encapsulation instances is initially discussed. Figure 22 illustrates the strongly dependent features of public methods. As depicted, Private-Variable stands out as one of the strongly dependent features of PublicMethod. Although Fig. 22 indicates an association between the increase in Public-Method and the increase in Private-Variable (indicating the necessity for Encapsulation), it is not adequate to conclude that all projects universally practice encapsulation. Additionally, Fig. 22 demonstrates that an increase in Public-Method is accompanied by increases in All-Methods and LOC. Moreover, All-Methods also rise with an increase in Concrete-Class instances. Consequently, the hypothesis gains better support by suggesting that, on average, the studied projects tend to practice encapsulation more as they grow in size.
- *Polymorphism*: Polymorphism, meaning “many forms”, is another crucial Object-Oriented Principle. Polymorphism has different types and forms such as Ad hoc polymorphism, Generics polymorphism, Overriding, and Overloading. The focus of this work would be on two common polymorphism types: Overriding and Overloading. Therefore, polymorphism is categorized into two distinct types: Overriding and Overloading.

  The first category, Overriding, involves a process where a subclass re-implements a method already present in the superclass. This ability enables software to offer different levels of functionality for a method within an inheritance chain. On the other hand, Overloading refers to the process of defining multiple methods with the same name in a single class, each serving a different purpose.

  Figure 20 showcases the features that strongly depend on Overloaded methods. The first feature is LOC, which serves as a conventional measure of software system size [16]. This suggests that, on average, developers tend to employ the

Overloading mechanism, and as the project's size increases, the number of overloaded methods also tends to rise. Conversely, Overridden methods exhibit a robust dependency on ClassInheritance and Super-methods, aligning with expectations, as illustrated in Fig. 21. However, there is not substantial evidence to support the idea that the Overriding mechanism is practiced more as software systems grow in size.

- *Inheritance*: Inheritance stands as one of the four core Object-Oriented Principles. It empowers classes to acquire attributes such as fields, methods, and behaviours from a shared superclass, fostering code reusability, modular design, and enhanced maintainability of the codebase. Through inheritance, classes can circumvent the need for redundant code, leading to more efficient development, simplified upkeep, and a more organized code structure. Figure 15 illustrates the features strongly dependent on the inheritance principle, encompassing both direct inheritance (e.g., super-class → sub-class, as depicted in Fig. 16) and indirect inheritance (e.g., super-class → super-class → sub-class). However, the features strongly reliant on inheritance do not provide definitive evidence of a correlation between inheritance and project size. Consequently, it can be concluded that larger projects do not necessarily adhere more to number of super-classes.
- *Commenting*: Nearly all programming languages incorporate a commenting mechanism that assists programmers in providing insights for the written code at various levels.

  Pooja et al. [17] delve into the roles of comments in Object-Oriented programming languages such as Java, Python, and Smalltalk, discussing aspects like Code Summary, Code Ownership, Code Usage, Deprecation details, Rationale behind the code, Development Notes, Parameters details, Code Intent, Collaborator, Examples, Class reference, Key implementation points, Key message, and more. In the case of Java, three distinct syntaxes for comments are available: 1) Line Comment; 2) Block Comment; 3) Java Doc. Typically, line comments suit brief comments requiring a line or less, block comments are ideal for more extensive comments needing greater detail, and Java doc is fitting for providing metainformation about the software, encompassing ownership, legal details, release updates, and related subjects.

  Figure 5 offers an overview of the strongly dependent features related to the trio of comment types. Notably, Block-Comment does not display strong dependence on any feature, including All-Comments, implying that it is not as widely used as Line-Comments and Java-Doc among programmers. Although Line-Comment is strongly reliant on All-Comments, it does not exhibit significant reliance on other features (see Fig. 7). In contrast, Java-Doc (Fig. 8) exhibits strong dependence on multiple features such as Overloaded-Methods, All-Methods, and Dead-Methods. This points to programmers' inclination to utilize Java-Doc to provide details about Overloaded-Methods, All-Methods, and Dead-Methods. Notably, the correlation between Java-Doc and Dead-Methods suggests programmers might employ Java-Doc to clarify outdated methods, though this conjecture necessitates further investigation. Since All-Methods is considered one of the measures of project size, an increase in project size consequently results in a higher usage of Java-Doc as well.
- *Declared Variable/Class Fields*: In this study, the class fields or variables at the class level are specifically focused, that exclude local variables (those declared within methods). The following sections will delve into various coding practices associated with variables:

1. Primitive Variables: Java encompasses various primitive variable types, including int, double, float, Boolean, char, byte, long, and short. As the name suggests, these variable types are primarily employed to store simple, non-complex fundamental data types, where the data are represented with a singular magnitude. Consequently, primitive data tend to occupy less memory compared to complex data (i.e., variables with class types).

   Figure 26 illustrates two features that exhibit strong dependence on Primitive Variables: 1) Static Variables and 2) Default Variables. Static Variables hold information about the class structure, rather than individual instances (objects), and are shared among all instances of the class. Due to their lasting presence in memory, proper management is crucial to avoid memory overflow. The strong correlation between Primitive Variables and Static Variables indicates the proficient utilization of Static Variables by programmers on average.

   The second feature strongly linked to Primitive Variables is Default Variables, which restricts the variable's visibility to the declaring class and all other classes within the same package. The reason for this dependency requires further analysis. However, this result does not provide evidence that the number of primitive variables increases as the project size grows.
2. Accessed Variables: Accessed Variables in this study refer to any non-primitive variable that is accessed by an identifier. The identifier can be: 1) the class name (when accessing a static variable from another class), 2) the object name (when accessing a non-static variable from either the same class or another class), or 3) "this" (used as an identifier when accessing a field from the same class).

   Figure 3 displays the strongly dependent features of Accessed Variables. The strongest dependent feature of Accessed Variables is Public Method. This observation could be linked to the encapsulation scenario, where Public Methods access variables from the same class (using the "this" identifier). This can be further supported by the fact that Private Variables are also strongly dependent features of Accessed Variables.
3. Final: Final Variables offer advantages in Object-Oriented programming, such as enhanced clarity and readability [18], [19], and in addition, they can substantially improve program performance, sometimes even up to 25 times higher [20].

Final-Variables behave like constant variables, meaning once a value is assigned to them, it remains unchanged. According to the findings in Fig. 13, programmers seem to be leaning towards increased usage of Final-Variables as the software system becomes larger. However, the exact intention behind this increase requires more research.

- *Exception Handling*: Modern programming languages are equipped with elaborate mechanisms to manage unforeseen situations that might arise due to various reasons, such as incompatible external inputs, limited resources, and poor-quality code structure. Java features an Exception-Handling mechanism that helps capture and address exceptions and errors. The Exception Handling mechanism in Java consists of four parts: 1) Try Block, 2) Catch Block, and 3) Finally Block. The last block is optional, while the first two are mandatory. The Try Block encapsulates the code susceptible to triggering an exception or error, while the Catch Block provides a handling solution. It is worth noting that most languages, including Java, allow a Try Block to have more than one Catch Block [21], indicating that the code within the Try Block might not be specific to just one vulnerability. This can be considered a code smell [22], [23].

  Figure 14 depicts the features dependent on exceptions. Each exception is permitted to have only one Try Block, Figure 14 illustrates the dependent features of the Try Block. The most significant feature dependent on the Try Block is the Catch Block, showing a dependency of over 97 %. This high dependency implies that only a very small portion of the entire Exception Handling structure utilizes multiple Catch Block, which could be indicative of a code smell. Furthermore, the Exception block demonstrates a strong dependency on Comments, possibly indicating programmers' tendencies to provide explanatory comments for Exception Blocks. Declared Variable and Overloaded Method are two other features exhibiting strong dependencies with Exception Handling.
- *Dead Code*: Dead code refers to any piece of code that would never be executed due to various reasons. Dead code can encompass methods/functions, classes, interfaces, or any blocks of code like loops or conditional statements. Code becomes dead when it remains uninvoked by other code or when it is situated under conditions that are impossible or incorrect (e.g., if(4%3 == 0)). A common cause of dead code arises when existing code is rewritten to alter its behaviour, effectively disconnecting the old code from the rest of the system. This phenomenon is often referred to as clone code [24]. In this study, our focus is exclusively on dead methods, which represent a subset of dead code. A method is considered dead if it satisfies the following criteria: 1) It is not invoked by any other method or constructor, and 2) It does not serve as a super method or an abstract method. Figure 12 illustrates the strongly dependent features of dead methods. As indicated, the most influential dependent feature of dead methods is All-Method, followed by Public-Method. The prominence of Public-Method among the features dependent on dead methods implies that public methods have undergone more changes compared to other methods (such as private, default, and protected methods). This suggests that insufficient attention might have been given to obsolete methods. In an ideal scenario, obsolete methods should be removed from the source code to prevent code cloning and maintenance confusion especially with the presence of version-control platforms such as Git. Another noteworthy finding is the correlation between dead methods and Doc Comments. This correlation can be interpreted as an attempt to provide comments for either the obsolete method itself or the newly introduced substitute.

B. Non-Conforming Features

Among all the features experimented with, as listed in Table II, a few features exhibit independent behaviour and appear to have random effects. These features are discussed in the following section:

- *Block-Comment*: Block comments do not exhibit a strong dependency on any particular feature. The feature most dependent on Block comments is All-comment, which demonstrates a Pearson Correlation Coefficient (PCC) of 67 %. This is followed by Line-comment and Doc-comment with PCC values of 56 % and 54 %, respectively. Considering that Line comments and Doc comments have previously shown strong dependencies on other features, it can be inferred that Block Comments are not the preferred choice of comment type among developers.
- *Abstract Class*: An abstract class is a mechanism that enables developers to create a class that combines both implemented and unimplemented elements, providing flexibility for future behaviours. While designed to enhance software system quality by curbing maintenance costs, it is noteworthy that the Abstract Class does not exhibit any dependency on other features. In other words, the presence of a larger number of abstract classes does not necessarily correlate with larger software systems.

  Furthermore, Abstract Classes do not display a dependency on the total number of classes either. This suggests that the utilization of Abstract classes is primarily dictated by specific use cases rather than systematic trends. The most dependent feature, Interface, displays a Pearson Correlation Coefficient (PCC) of 44 %, yet it does not indicate a strong dependency on the Abstract class.
- *Interface*: The interface is another mechanism that holds the potential to enhance the maintainability of a software system. The primary distinction between an abstract class and an interface lies in the fact that an interface can solely contain unimplemented methods, also known as signature methods. Interestingly, an interface does not exhibit a strong dependency on any particular feature. However, the feature most dependent on an interface is the Inner-Class, displaying a Pearson Correlation Coefficient (PCC) of 66 %.

  A plausible explanation for this moderate dependency could be that classes inheriting from an interface, or multiple interfaces, are, to some extent, tied to the behaviour outlined by those interface(s). The use of inner classes could provide a means to compensate such ties and introduce alternative types of behaviour within the class [25].

- *Final Method*: The final modifier applied to methods indicates that the method cannot be overridden if the owning class is inherited by another class. This feature, Final Method, does not demonstrate a strong dependency on any other feature. However, it does exhibit an average dependency with the following features: 1) LOC, 2) Protected Method, 3) Overloaded Method, 4) All-Comment, and 5) Variable-Final.

  The Line of Code (LOC) is a versatile feature that can easily demonstrate dependencies with numerous other features. Among the features dependent on Final Method, Protected Method shows the highest level of dependency. While further investigation is necessary to ascertain the nature of this dependency, the average relationship between Final Method and Protected Method can be understood as developers prioritizing heightened source code security, particularly when both mechanisms are appropriately utilized.

  The average dependency between Final Method and Overloaded Method could be attributed to programmers' endeavours to safeguard methods with similar names from accidental override in the future. The dependence of All-Comment on Final Method could signify developers' inclination to provide detailed explanations for their chosen modifiers. The average dependency with Variable-Final indicates a similar trend.
- *Default Method*: The default modifier applied to methods (note that the default modifier itself has no keyword) restricts the visibility of methods to within their package. This feature, Default Method, does not exhibit a strong dependency on any other feature. The most significant feature associated with Default Method is InnerClass, showing a Pearson Correlation Coefficient (PCC) of 50 %. However, this coefficient does not point to a substantial connection between Default Method and any other feature.
- *Private Method*: The private method is a feature that demonstrates a moderately strong dependency with All-Methods (PCC 69 %), DeclaredVariable (PCC 68 %), and Variable-Final (PCC 66 %). The interrelation between Private Method and All-Methods suggests a notably significant utilization of private methods by developers. Since private methods are accessible only within their respective class containers, the relatively high dependency between Private Method and Declared-Variable might be attributed to developers' efforts to handle data internally within the class through private methods.
- *Protected Methods*: Protected methods share similarities with default methods, but they come with a distinction: protected methods can be accessed by all subclasses of the defining class. Protected methods do not demonstrate any notable dependency on other features. However, a relatively strong dependency exists between Protected Methods and LOC (PCC 69 %), All-Comments (PCC 66 %), and Declared-Variable (PCC 65 %).
- *Static Method*: Static methods are categorized as class methods rather than object methods. This implies that there is precisely one instance of any static method, which can be accessed via the class itself, and not necessarily through an object. The most significant dependencies of Static Method are Variable-Primitive (PCC 60 %) and Variable-Static (PCC 55 %). This pattern can be understood as developers' intention to execute auxiliary processes using static methods in conjunction with static and primitive variables.
- *Variable-Protect*: The protected variable stands out as one of the features that lacks a discernible pattern in relation to other features. The most notable feature dependent on Protected Variable is Doc-Comment, with a Pearson Correlation Coefficient (PCC) of 29 %. However, this coefficient does not suggest any substantial or meaningful connection.
- *Variable-Public*: Public variables exhibit an average level of dependency on several other features, including Declared-Variable (PCC 65 %), Variable-Final (PCC 59 %), and Public-Method (PCC 57 %). The 65 % Pearson Correlation Coefficient (PCC) between Public Variable and Declared-Variable can be attributed to the relatively high frequency of public variables in comparison with other visibility modifiers. Additionally, the average dependence of Final Variables on Public Variables might suggest that on average Final Variables are also declared as public.
- *Average Method and Class LOC*: In addition to the overall Line of Code (LOC), this study evaluates two other features: the average Line of Code across all methods and the average Line of Code across all classes and interfaces. The average method LOC does not exhibit noteworthy dependency on any feature, with the highest correlation being with Variable-Default (PCC 39 %).

  The average class LOC also does not show significant dependency on any other feature. However, the features most dependent on average class LOC are LOC (PCC 65 %), Doc-Comment (PCC 62 %), and Exception (PCC 61 %).
- *Package-Count*: Another feature displaying no strong dependency on any other feature is Package-Count. The feature most dependent on Package-Count is Concrete Classes with a Pearson Correlation Coefficient (PCC) of 61 %. While this dependency is not robust, it aligns with expectations. Given that Package-Count exhibits no significant dependence on any other feature, it can be inferred that packaging plays a relatively minor role in developers' architectural decisions.

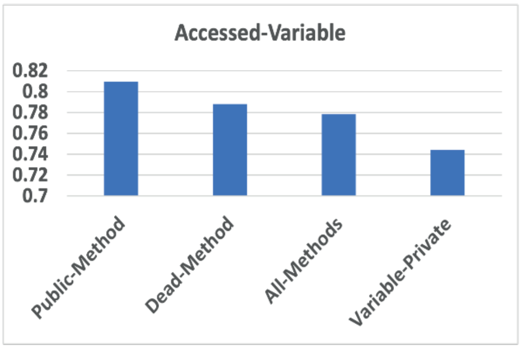


Fig. 3. Accessed-Variables details.

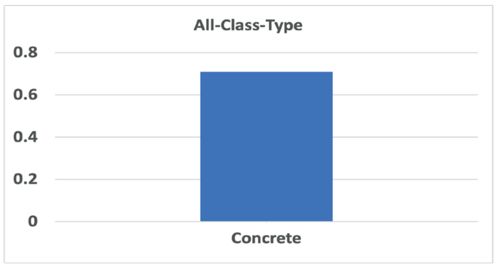


Fig. 4. All-Class-Type details.

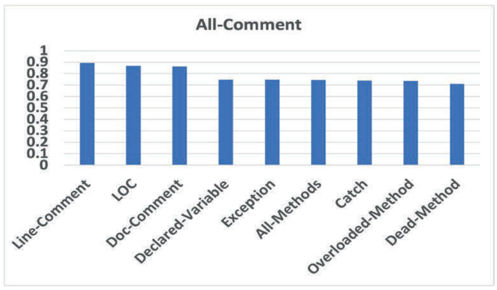


Fig. 5. All-Comments details.

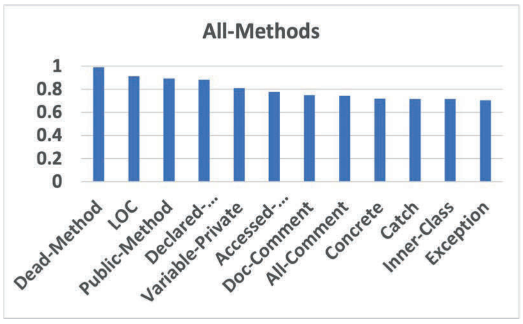


Fig. 6. All-Methods details.

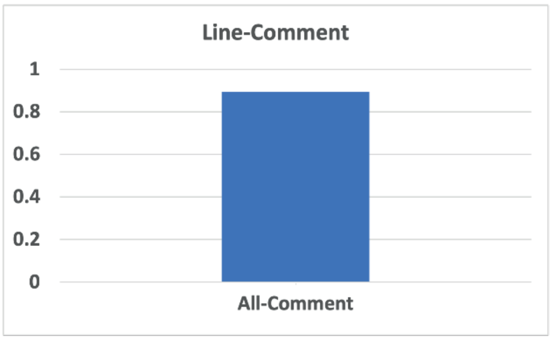


Fig. 7. Line-Comments details.

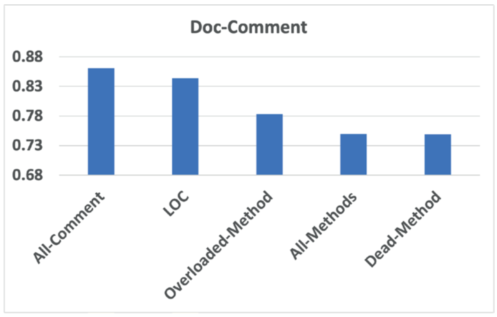


Fig. 8. Doc-Comments details.

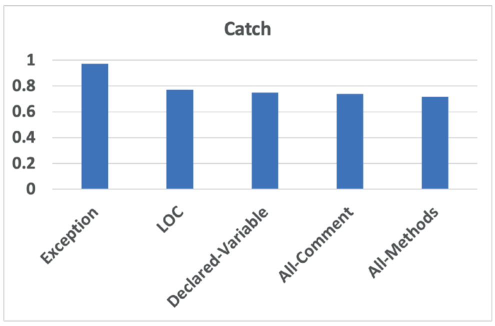


Fig. 9. Catch details.

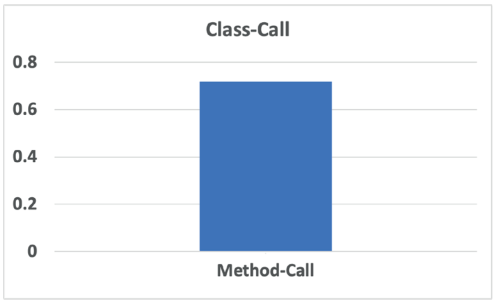


Fig. 10. Class-Call details.

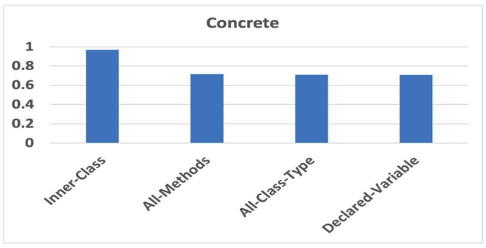


Fig. 11. Concrete details.

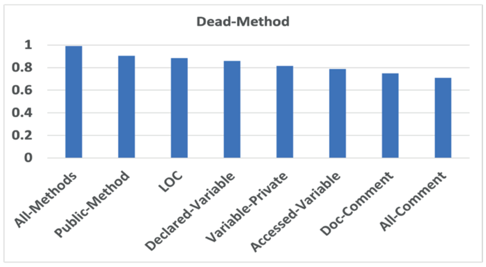


Fig. 12. Dead-Method details.

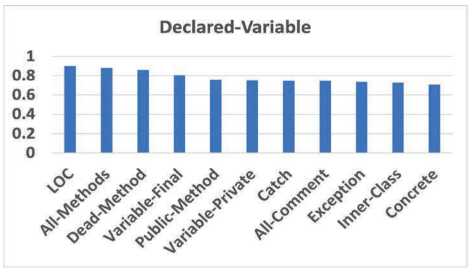


Fig. 13. Declared-Variable details.

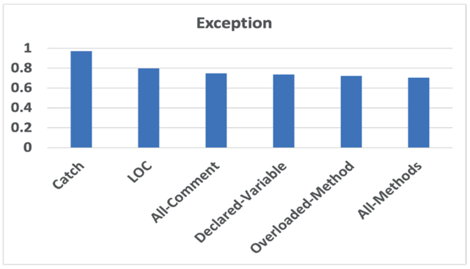


Fig. 14. Exception details.

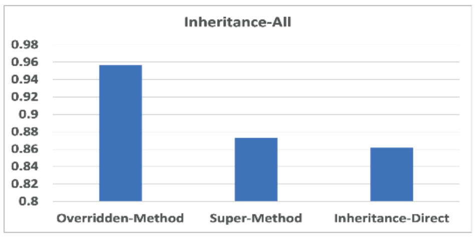


Fig. 15. Inheritance-All details.

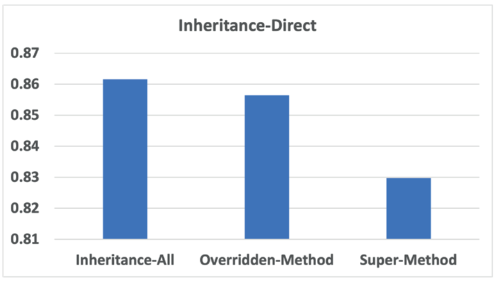


Fig. 16. Inheritance-Direct details.

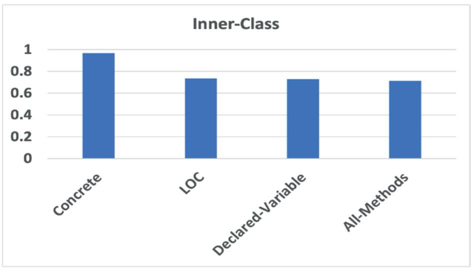


Fig. 17. Inner-Classes details.

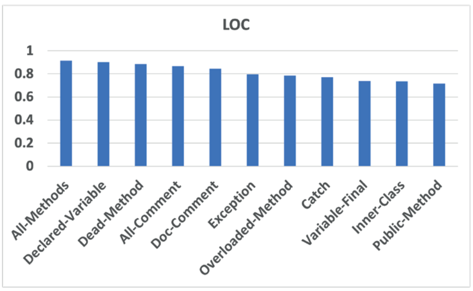


Fig. 18. LOC details.

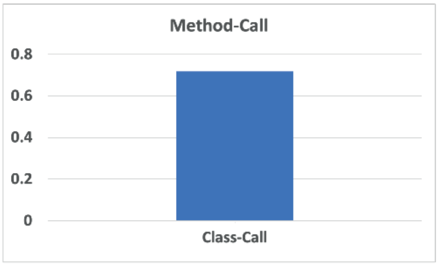


Fig. 19. Method-Call details.

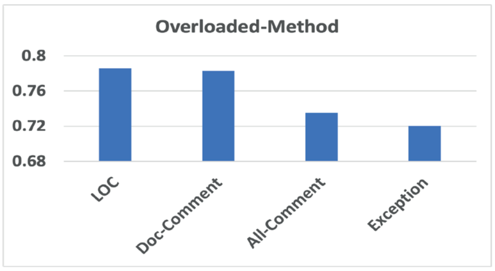


Fig. 20. Overloaded-Methods details.

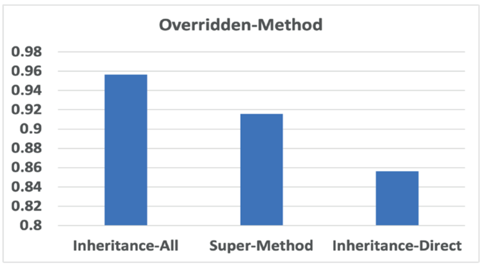


Fig. 21. Overridden-Methods details.

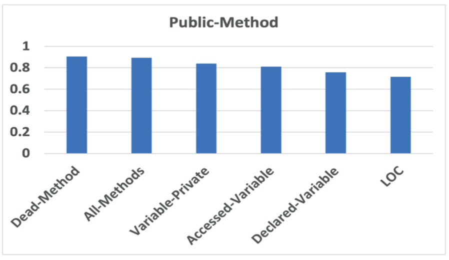


Fig. 22. Public-Method details.

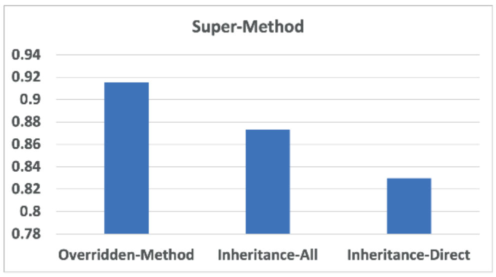


Fig. 23. Super-Method details.

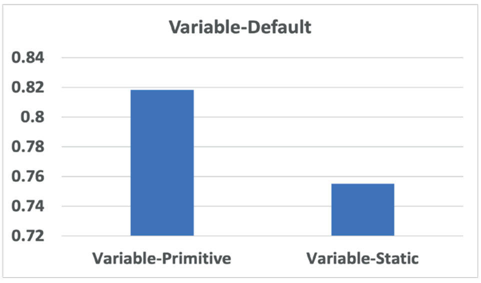


Fig. 24. Variable-Default details.

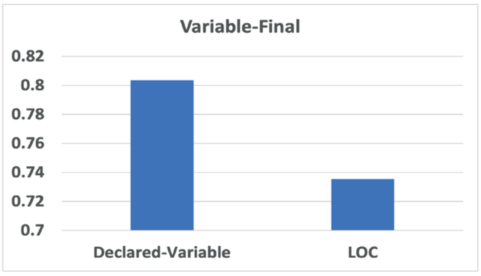


Fig. 25. Variable-Final details.

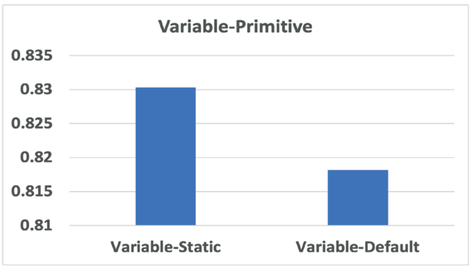


Fig. 26. Variable-Primitive details.

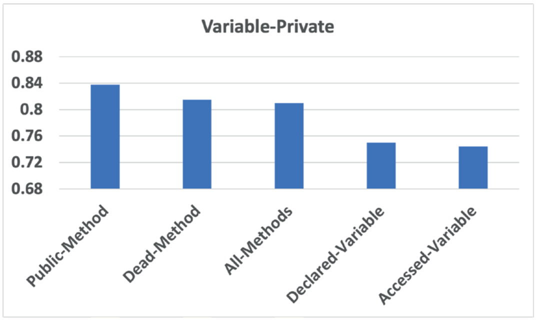


Fig. 27. Variable-Private details.

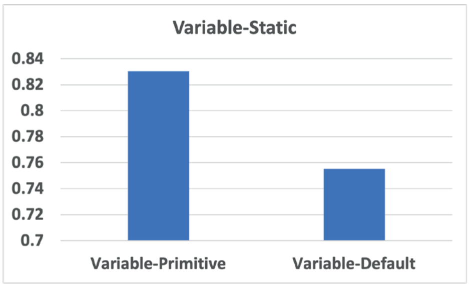


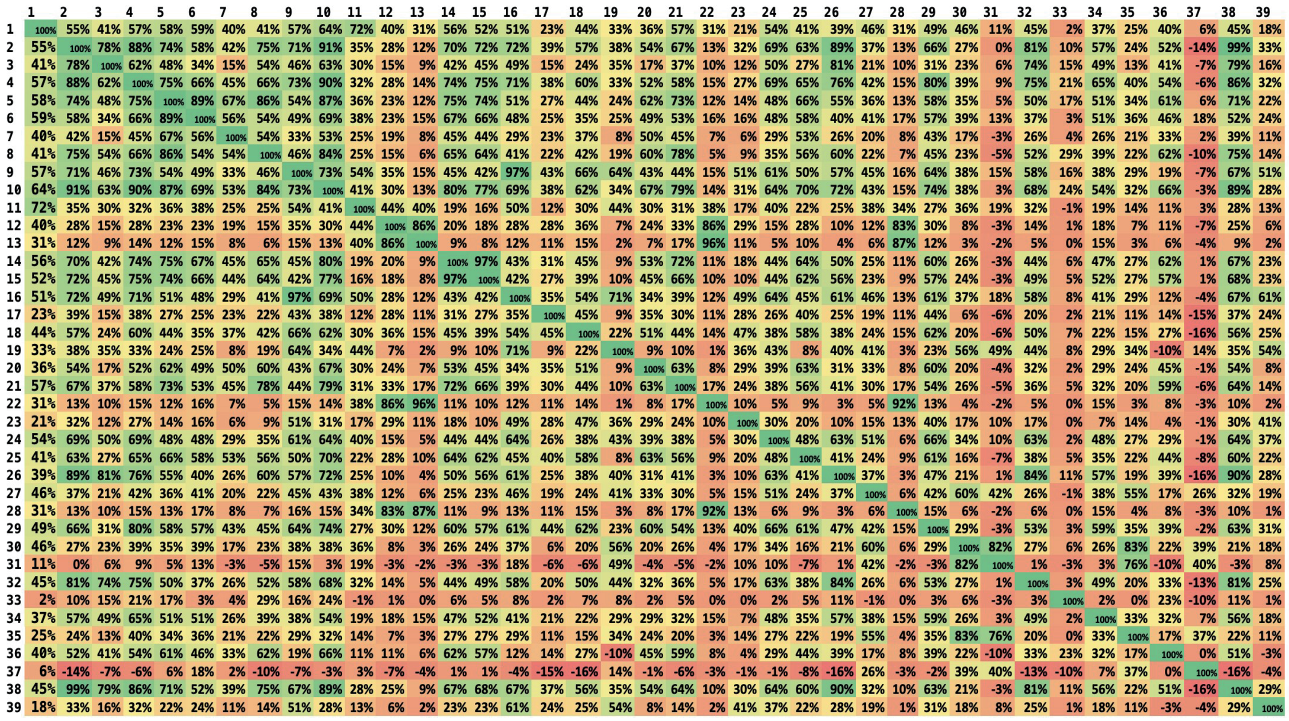
Fig. 28. Variable-Static details.

IV. Conclusion

In this study, a statistical analysis of a collection of Java software projects (each containing a minimum of 150 Java files) has been conducted. This analytical work involves targeting and computing various features (39 features in total, as outlined in Table II) for each project. The primary objective of this study has been to unveil the interdependencies among source code features, aiming to uncover structural intricacies within the source code.

Figure 29 provides a comprehensive view of the dependencies among all the features (Note that all values are colour-coded from the highest – dark green to the lowest – dark red). Those features displaying strong dependencies are discussed in Figs. 3–28. The analysis results suggest that developers, on average, adhere to the requisite practices for implementing Object-Oriented Principles (excluding Abstraction, which cannot be assessed using the designated methodology). Furthermore, the analysis reveals that Line Comments are the most frequently employed type of comments among programmers. However, the utilization of Block Comments does not exhibit any discernible pattern.

On average, developers have made efforts to evade a particular code smell associated with Exception Handling, notably, the use of multiple Catch blocks for a single Try block. Interestingly, an increase in the size of method bodies and class bodies does not necessarily correlate with an increase in other features. Another noteworthy observation pertains to the utilization of Interfaces and Abstract Classes, which does not see a proportional rise with project size. This holds true, even more remarkably, for Abstract Classes. It is worth noting that there is no average, strong, or very strong negative dependency between any pair of features. The result of this study, in general, may assist developers in monitoring the evolution of the project. A Java software system is a complex and multidimensional entity where, in large systems, maintaining the quality of the code at a satisfactory level becomes challenging. The findings of this paper provide an overview of how one feature of the source code increases or decreases in relation to another. For instance, the general finding shows that the number of dead methods increases with the increase of public methods. Furthermore, the findings of this work reveal the importance of utilizing a static code analyser to observe changes over the course of development for any in-vivo software system.



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39
1	100%	55%	41%	57%	58%	59%	40%	41%	57%	64%	72%	40%	31%	56%	52%	51%	23%	44%	33%	36%	57%	31%	21%	54%	41%	39%	46%	31%	49%	46%	11%	45%	2%	37%	25%	40%	6%	45%	18%
2	55%	100%	78%	88%	74%	58%	42%	75%	71%	91%	35%	28%	12%	70%	72%	72%	39%	57%	38%	54%	67%	13%	32%	69%	63%	89%	37%	13%	66%	27%	0%	81%	10%	57%	24%	52%	-14%	99%	33%
3	41%	78%	100%	62%	48%	34%	15%	54%	46%	63%	30%	15%	9%	42%	45%	49%	15%	24%	35%	17%	37%	10%	12%	50%	27%	81%	21%	10%	31%	23%	6%	74%	15%	49%	13%	41%	-7%	79%	16%
4	57%	88%	62%	100%	75%	66%	45%	66%	73%	90%	32%	28%	14%	74%	75%	71%	38%	60%	33%	52%	58%	15%	27%	69%	65%	76%	42%	15%	80%	39%	9%	75%	21%	65%	40%	54%	-6%	86%	32%
5	58%	74%	48%	75%	100%	89%	67%	86%	54%	87%	36%	23%	12%	75%	74%	51%	27%	44%	24%	62%	73%	12%	14%	48%	66%	55%	36%	13%	58%	35%	5%	50%	17%	51%	34%	61%	6%	71%	22%
6	59%	58%	34%	66%	89%	100%	56%	54%	49%	69%	38%	23%	15%	67%	66%	48%	25%	35%	25%	49%	53%	16%	16%	48%	58%	40%	41%	17%	57%	39%	13%	37%	3%	51%	36%	46%	18%	52%	24%
7	40%	42%	15%	45%	67%	56%	100%	54%	33%	53%	25%	19%	8%	45%	44%	29%	23%	37%	8%	50%	45%	7%	6%	29%	53%	26%	20%	8%	43%	17%	-3%	26%	4%	26%	21%	33%	2%	39%	11%
8	41%	75%	54%	66%	86%	54%	54%	100%	46%	84%	25%	15%	6%	65%	64%	41%	22%	42%	19%	60%	78%	5%	9%	35%	56%	60%	22%	7%	45%	23%	-5%	52%	29%	39%	22%	62%	-10%	75%	14%
9	57%	71%	46%	73%	54%	49%	33%	46%	100%	73%	54%	35%	15%	45%	42%	97%	43%	66%	64%	43%	44%	15%	51%	61%	50%	57%	45%	16%	64%	38%	15%	58%	16%	38%	29%	19%	-7%	67%	51%
10	64%	91%	63%	90%	87%	69%	53%	84%	73%	100%	41%	30%	13%	80%	77%	69%	38%	62%	34%	67%	79%	14%	31%	64%	70%	72%	43%	15%	74%	38%	3%	68%	24%	54%	32%	66%	-3%	89%	28%
11	72%	35%	30%	32%	36%	38%	25%	25%	54%	41%	100%	44%	40%	19%	16%	50%	12%	30%	44%	30%	31%	38%	17%	40%	22%	25%	38%	34%	27%	36%	19%	32%	-1%	19%	14%	11%	3%	28%	13%
12	40%	28%	15%	28%	23%	23%	19%	15%	35%	30%	44%	100%	86%	20%	18%	28%	28%	36%	7%	24%	33%	86%	29%	15%	28%	10%	12%	83%	30%	8%	-3%	14%	1%	18%	7%	11%	-7%	25%	6%
13	31%	12%	9%	14%	12%	15%	8%	6%	15%	13%	40%	86%	100%	9%	8%	12%	11%	15%	2%	7%	17%	96%	11%	5%	10%	4%	6%	87%	12%	3%	-2%	5%	0%	15%	3%	6%	-4%	9%	2%
14	56%	70%	42%	74%	75%	67%	45%	65%	45%	80%	19%	20%	9%	100%	97%	43%	31%	45%	9%	53%	72%	11%	18%	44%	64%	50%	25%	11%	60%	26%	-3%	44%	6%	47%	27%	62%	1%	67%	23%
15	52%	72%	45%	75%	74%	66%	44%	64%	42%	77%	16%	18%	8%	97%	100%	42%	27%	39%	10%	45%	66%	10%	10%	44%	62%	56%	23%	9%	57%	24%	-3%	49%	5%	52%	27%	57%	1%	68%	23%
16	51%	72%	49%	71%	51%	48%	29%	41%	97%	69%	50%	28%	12%	43%	42%	100%	35%	54%	71%	34%	39%	12%	49%	64%	45%	61%	46%	13%	61%	37%	18%	58%	8%	41%	29%	12%	-4%	67%	61%
17	23%	39%	15%	38%	27%	25%	23%	22%	43%	38%	12%	28%	11%	31%	27%	35%	100%	45%	9%	35%	30%	11%	28%	26%	40%	25%	19%	11%	44%	6%	-6%	20%	2%	21%	11%	14%	-15%	37%	24%
18	44%	57%	24%	60%	44%	35%	37%	42%	66%	62%	30%	36%	15%	45%	39%	54%	45%	100%	22%	51%	44%	14%	47%	38%	58%	38%	24%	15%	62%	20%	-6%	50%	7%	22%	15%	27%	-16%	56%	25%
19	33%	38%	35%	33%	24%	25%	8%	19%	64%	34%	44%	7%	2%	9%	10%	71%	9%	22%	100%	9%	10%	1%	36%	43%	8%	40%	41%	3%	23%	56%	49%	44%	8%	29%	34%	-10%	14%	35%	54%
20	36%	54%	17%	52%	62%	49%	50%	60%	43%	67%	30%	24%	7%	53%	45%	34%	35%	51%	9%	100%	63%	8%	29%	39%	63%	31%	33%	8%	60%	20%	-4%	32%	2%	29%	24%	45%	-1%	54%	8%
21	57%	67%	37%	58%	73%	53%	45%	78%	44%	79%	31%	33%	17%	72%	66%	39%	30%	44%	10%	63%	100%	17%	24%	38%	56%	41%	30%	17%	54%	26%	-5%	36%	5%	32%	20%	59%	-6%	64%	14%
22	31%	13%	10%	15%	12%	16%	7%	5%	15%	14%	38%	86%	96%	11%	10%	12%	11%	14%	1%	8%	17%	100%	10%	5%	9%	3%	5%	92%	13%	4%	-2%	5%	0%	15%	3%	8%	-3%	10%	2%
23	21%	32%	12%	27%	14%	16%	6%	9%	51%	31%	17%	29%	11%	18%	10%	49%	28%	47%	36%	29%	24%	10%	100%	30%	20%	10%	15%	13%	40%	17%	10%	17%	0%	7%	14%	4%	-1%	30%	41%
24	54%	69%	50%	69%	48%	48%	29%	35%	61%	64%	40%	15%	5%	44%	44%	64%	26%	38%	43%	39%	38%	5%	30%	100%	48%	63%	51%	6%	66%	34%	10%	63%	2%	48%	27%	29%	-1%	64%	37%
25	41%	63%	27%	65%	66%	58%	53%	56%	50%	70%	22%	28%	10%	64%	62%	45%	40%	58%	8%	63%	56%	9%	20%	48%	100%	41%	24%	9%	61%	16%	-7%	38%	5%	35%	22%	44%	-8%	60%	22%
26	39%	89%	81%	76%	55%	40%	26%	60%	57%	72%	25%	10%	4%	50%	56%	61%	25%	38%	40%	31%	41%	3%	10%	63%	41%	100%	37%	3%	47%	21%	1%	84%	11%	57%	19%	39%	-16%	90%	28%
27	46%	37%	21%	42%	36%	41%	20%	22%	45%	43%	38%	12%	6%	25%	23%	46%	19%	24%	41%	33%	30%	5%	15%	51%	24%	37%	100%	6%	42%	60%	42%	26%	-1%	38%	55%	17%	26%	32%	19%
28	31%	13%	10%	15%	13%	17%	8%	7%	16%	15%	34%	83%	87%	11%	9%	13%	11%	15%	3%	8%	17%	92%	13%	6%	9%	3%	6%	100%	15%	6%	-2%	6%	0%	15%	4%	8%	-3%	10%	1%
29	49%	66%	31%	80%	58%	57%	43%	45%	64%	74%	27%	30%	12%	60%	57%	61%	44%	62%	23%	60%	54%	13%	40%	66%	61%	47%	42%	15%	100%	29%	-3%	53%	3%	59%	35%	39%	-2%	63%	31%
30	46%	27%	23%	39%	35%	39%	17%	23%	38%	38%	36%	8%	3%	26%	24%	37%	6%	20%	56%	20%	26%	4%	17%	34%	16%	21%	60%	6%	29%	100%	82%	27%	6%	26%	83%	22%	39%	21%	18%
31	11%	0%	6%	9%	5%	13%	-3%	-5%	15%	3%	19%	-3%	-2%	-3%	-3%	18%	-6%	-6%	49%	-4%	-5%	-2%	10%	10%	-7%	1%	42%	-2%	-3%	82%	100%	1%	-3%	3%	76%	-10%	40%	-3%	8%
32	45%	81%	74%	75%	50%	37%	26%	52%	58%	68%	32%	14%	5%	44%	49%	58%	20%	50%	44%	32%	36%	5%	17%	63%	38%	84%	26%	6%	53%	27%	1%	100%	3%	49%	20%	33%	-13%	81%	25%
33	2%	10%	15%	21%	17%	3%	4%	29%	16%	24%	-1%	1%	0%	6%	5%	8%	2%	7%	8%	2%	5%	0%	0%	2%	5%	11%	-1%	0%	3%	6%	-3%	3%	100%	2%	0%	23%	-10%	11%	1%
34	37%	57%	49%	65%	51%	51%	26%	39%	38%	54%	19%	18%	15%	47%	52%	41%	21%	22%	29%	29%	32%	15%	7%	48%	35%	57%	38%	15%	59%	26%	3%	49%	2%	100%	33%	32%	7%	56%	18%
35	25%	24%	13%	40%	34%	36%	21%	22%	29%	32%	14%	7%	3%	27%	27%	29%	11%	15%	34%	24%	20%	3%	14%	27%	22%	19%	55%	4%	35%	83%	76%	20%	0%	33%	100%	17%	37%	22%	11%
36	40%	52%	41%	54%	61%	46%	33%	62%	19%	66%	11%	11%	6%	62%	57%	12%	14%	27%	-10%	45%	59%	8%	4%	29%	44%	39%	17%	8%	39%	22%	-10%	33%	23%	32%	17%	100%	0%	51%	-3%
37	6%	-14%	-7%	-6%	6%	18%	2%	-10%	-7%	-3%	3%	-7%	-4%	1%	1%	-4%	-15%	-16%	14%	-1%	-6%	-3%	-1%	-1%	-8%	-16%	26%	-3%	-2%	39%	40%	-13%	-10%	7%	37%	0%	100%	-16%	-4%
38	45%	99%	79%	86%	71%	52%	39%	75%	67%	89%	28%	25%	9%	67%	68%	67%	37%	56%	35%	54%	64%	10%	30%	64%	60%	90%	32%	10%	63%	21%	-3%	81%	11%	56%	22%	51%	-16%	100%	29%
39	18%	33%	16%	32%	22%	24%	11%	14%	51%	28%	13%	6%	2%	23%	23%	61%	24%	25%	54%	8%	14%	2%	41%	37%	22%	28%	19%	1%	31%	18%	8%	25%	1%	18%	11%	-3%	-4%	29%	100%

Fig. 29. The complete inter-dependency of all 39 features with each other.

V. Future Work

This study has been based solely on the statistical characteristics of a few hundred open-source Java projects. Consequently, it might uncover only specific types of details. As part of future endeavours, more in-depth analyses of software projects could be conducted to unveil finer-grained structural intricacies.

References

[1] C. De Souza, J. Froehlich, and P. Dourish, "Seeking the source: software source code as a social and technical artifact," in *Proceedings of the 2005 International ACM SIGGROUP Conference on Supporting Group Work*, Nov. 2005, pp. 197–206. https://doi.org/10.1145/1099203.1099239

[2] K. Sharan and A. L. Davis, *Beginning Java 17 Fundamentals: Object-Oriented Programming in Java 17*, Springer, 2021.

[3] M. Allamanis and C. Sutton, "Mining source code repositories at massive scale using language modeling," in *2013 10th Working Conference on Mining Software Repositories (MSR)*, San Francisco, CA, USA, May 2013, pp. 207–216. https://doi.org/10.1109/MSR.2013.6624029

[4] A. Marcus and J. I. Maletic, "Identification of high-level concept clones in source code," in *Proceedings 16th annual international conference on automated software engineering (ASE 2001)*, San Diego, CA, USA, Nov. 2001, pp. 107–114. https://doi.org/10.1109/ASE.2001.989796

[5] P. Zeng, G. Lin, J. Zhang, and Y. Zhang, "Intelligent detection of vulnerable functions in software through neural embedding-based code analysis," *International Journal of Network Management*, vol. 33, no. 3, 2023, Art. no. e2198. https://doi.org/10.1002/nem.2198

[6] A. Puspaningrum, M. A. A. Hilmi, M. Mustamiin, M. I. Ginanjar, and Darsih, "Vulnerable source code detection using sonarcloud code analysis," *arXiv*, preprint arXiv:2307.02446, 2023. https://doi.org/10.48550/arXiv.2307.02446

[7] A. Kaur and R. Nayyar, "A comparative study of static code analysis tools for vulnerability detection in C/C++ and Java source code," *Procedia Computer Science*, vol. 171, 2020, pp. 2023–2029. https://doi.org/10.1016/j.procs.2020.04.217

[8] J. Oliveira, M. Souza, M. Flauzino, R. Durelli, and E. Figueiredo, "Can source code analysis indicate programming skills? A survey with developers," in *International Conference on the Quality of Information and Communications Technology*, Sep. 2022, pp. 156–171. https://doi.org/10.1007/978-3-031-14179-9_11

[9] I. Ruiz-Rube, T. Person, J. M. Dodero, J. M. Mota, J. M. Sánchez-Jara, "Applying static code analysis for domain-specific languages," *Software and Systems Modeling*, vol. 19, no. 1, Apr. 2020, pp. 95–110. https://doi.org/10.1007/s10270-019-00729-w

[10] D. Giebas and R. Wojszczyk, "Detection of concurrency errors in multithreaded applications based on static source code analysis," *IEEE Access*, vol. 9, pp. 61298–61323, Apr. 2021. https://doi.org/10.1109/ACCESS.2021.3073859

[11] K. Kawaguchi, "Java API for GitHub." [Online]. Available: https://github.com/hub4j/github-api. Accessed on: Aug. 13, 2023.

[12] I. Cohen, Y. Huang, J. Chen, J. Benesty, J. Benesty, J. Chen, Y. Huang, and I. Cohen, "Pearson correlation coefficient," in *Noise Reduction in Speech Processing. Springer Topics in Signal Processing*, vol. 2. Springer, Berlin, Heidelberg, 2009, pp. 1–4. https://doi.org/10.1007/978-3-642-00296-0_5

[13] P. Schober, C. Boer, and L. A. Schwarte, "Correlation coefficients: appropriate use and interpretation," *Anesthesia & Analgesia*, vol. 126, no. 5, May 2018, pp. 1763–1768. https://doi.org/10.1213/ANE.0000000000002864

[14] K. Bhatt, V. Tarey, P. Patel, K. B. Mits, and D. Ujjain, "Analysis of source lines of code (SLOC) metric," *International Journal of Emerging Technology and Advanced Engineering*, vol. 2, no. 5, May 2012, pp. 150–154. https://www.researchgate.net/profile/Kaushal-Bhatt-5/publication/281840565_Analysis_Of_Source_Lines_Of_CodeSLOC_Metric/links/55fab79608aeba1d9f37bcac/Analysis-Of-Source-Lines-Of-CodeSLOC-Metric.pdf

[15] E. Morozoff, "Using a line of code metric to understand software rework," *IEEE Software*, vol. 27, no. 1, Sep. 2009, pp. 72–77. https://doi.org/10.1109/MS.2009.160

[16] R. Park, "Software size measurement: A framework for counting source statements," Tech. Rep. CMU/SEI-92-TR-020, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 1992. [Online]. Available: https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=d3f99e79a92ba536f90ffae0a1272424751ae6ea

[17] P. Rani, S. Panichella, M. Leuenberger, A. Di Sorbo, and O. Nierstrasz, "How to identify class comment types? A multi-language approach for class comment classification," *Journal of Systems and Software*, vol. 181, 2021, Art. no. 111047. https://doi.org/10.1016/j.jss.2021.111047

[18] D. Greenfieldboyce and J. S. Foster, "Type qualifier inference for Java," in *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Objectoriented Programming Systems, Languages and Applications*, Oct. 2007, pp. 321–336. https://doi.org/10.1145/1297027.1297051

[19] M. Coblenz, J. Sunshine, J. Aldrich, B. Myers, S. Weber, and F. Shull, "Exploring language support for immutability," in *Proceedings of the 38th International Conference on Software Engineering*, May 2016, pp. 736–747. https://doi.org/10.1145/2884781.2884798

[20] D. Strmecki and D. Strmecki, "The Java final keyword – impact on performance – baeldung," May 2021. [Online]. Available: https://www.baeldung.com/java-final-performance

[21] S. Nakshatri, M. Hegde, and S. Thandra, "Analysis of exception handling patterns in Java projects: An empirical study," in *Proceedings of the 13th International Conference on Mining Software Repositories*, May 2016, pp. 500–503. https://doi.org/10.1145/2901739.2903499

[22] S. Tarwani and A. Chug, "Illustration and detection of exception handling bad smells," in *2021 8th International Conference on Computing for Sustainable Global Development (INDIACom)*, New Delhi, India, Jun. 2021, pp. 804–810. https://ieeexplore.ieee.org/document/9441470

[23] R. C. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship*, Pearson Education, 2009.

[24] C. Na, Y. Choi, J.-H. Lee, "DIP: Dead code insertion based black-box attack for programming language model," in *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics*, vol. 1, Toronto, Canada, Jul. 2023, pp. 7777–7791. https://doi.org/10.18653/v1/2023.acl-long.430

[25] Y. Smaragdakis, "Interfaces for nested classes," in *The 8th International Workshop on ObjectOriented Languages (FOOL8)*, 2001. [Online]. Available: https://www.cis.upenn.edu/~bcpierce/FOOL/FOOL8/yannis.pdf

Farshad Ghassemi Toosi earned his PhD in Computer Science from the University of Limerick in Ireland in 2017. His doctoral research focused on data visualization, particularly in the domain of Graph Drawing, and the application of Genetic Algorithms in this field. Following the completion of his PhD, he served as a Post-Doctorate for approximately two years, concentrating on Software Engineering with a specific emphasis on Source Code Manipulation and Feature Location. Subsequently, he assumed the role of a full-time Lecturer at the Department of Computer Science of Munster Technological University in Cork starting from 2019. Farshad Ghassemi Toosi is an esteemed academic member of Lero, the SFI Research Centre for Software in Ireland.

E-mail: farshad.toosi@mtu.ie
ORCID iD: https://orcid.org/0000-0002-1105-4819