Method level static source code analysis on behavioral change impact analysis in software regression testing

Fredrick Mugambi Muthengi[1], David Muchangi Mugo[1], Stephen Makau Mutua[2], Faith Mueni Musyoka[1]
[1]Department of Computing and Information Technology, Faculty of Applied Sciences, University of Embu, Embu, Kenya
[2]Department of Computer Science, Meru University of Science and Technology, Meru, Kenya

Article Info

Article history:

Received Feb 8, 2024
Revised Mar 16, 2024
Accepted Mar 28, 2024

Keywords:

Affected method
Associated method
Changed method
Regression testing
Source code analysis

ABSTRACT

Though a myriad of changes take place in a software system during maintenance, behavioral changes carry the bulk of the reasons for software modifications. In assessing the impact of the changes made in the software, static source code analysis can be a little complex depending on the reason for the expedition. Despite the works done so far, little focus has been directed on the potential of changed methods during static source code analysis, in assessing the impact of the changes made in software. This study investigates a method-level static source code analysis technique that would generate information on the methods affected by changes made in the software. The work analyzed three Java projects. The results indicate an improvement in leveraging on the knowledge of edited methods in change impact assessment during regression testing. The approach enhances code review efforts in light of assessing operational behavior impacted by the changes made.

Corresponding Author:

Fredrick Mugambi Muthengi
Department of Computing and Information Technology, Faculty of Applied Sciences, University of Embu
Embu, Kenya
Email: fremugambi@gmail.com

1. INTRODUCTION

Static source code analysis is a quality assurance measure used in examining a program's source code without executing the program [1]. The goal of the various static source code analysis tools varies based on the objective of analysis they are expected to perform [2] and the programming languages they support [3]. Among the objectives of static source code analysis, first is control flow analysis, that examines the sequence of instructions executed in a program to identify potential defects [4]. Second, data flow analysis that examines how data is flowing through a program to identify faults such as uninitialized variables, null pointers, and buffer overflows [4]–[7]. Third, code review that involves a manual inspection of the source code by a team of expert programmers to unearth logic errors, syntax flaws, and security vulnerabilities among others [8], [9]. Fourth, code metrics analysis measuring various characteristics of the code such as complexity, coupling, and cohesion [10], [11]. The fifth approach proposed in previous research is pattern-matching techniques that examines the code to identify issues like violation to coding standards and best-practices, that have the potential of introducing defects in the code [12]–[14]. All these five approaches have one thing in common: identification of potential defects in the program.

Although static source code analysis has received significant attention for decades, there is little empirical evidence on method-level analysis geared towards assessing the impact of the changes made on the software. From the literature searches, no research has applied the method-level static source code analysis in assessing the impact of the changes made to a software's operational behavior. It is this motivation that compels this study to investigate the possible usage of method-level static source code analysis in revealing method-level change knowledge that would expose the system's behavior affected by the changes made in a software system.

Method-level source code analysis is a variant of targeted source code analysis approach. Using targeted code analysis [15], code sections that require special attention can be identified and a retrieval of key code targets information [16] can ease the process of mapping tests to target code sections. Alterations in a code block that has been modified can also be uncovered [17]. At the method level, an analysis of the changes made to individual methods is done to assess the quality of the changes made in the system [18]. Recent research shows a significant increase into the development of automated tools for static source code analysis like FindBugs [3] and extract method feature [19]. Both extract method refactoring and composite method refactoring [20] tools, help improve program understandability and comprehension supported by an empirical study on the characteristics of method extractions in Java [21].

Software developers prefer source code information on method levels in code analysis tasks [22]. CodeTracker in [23] presents an approach of tracking the history of software code commits by generating commit histories for methods and variables. Its usage is in the raw examination of commit histories for given methods or variables. The user has to input the method or variable of interest. A recent study by [22] explored the practicality of studying the changes made in a method laying the more emphasis on the value of changed methods in light of examining the overall impact of changes made on the system's functionality. However, their approach was based on the user's input of a particular method to study the changes that have taken place on a given method after several code commits affecting the method.

Other than focusing on expertscode analysis issues, upcoming industry practitioners on trainings for writing good computer programs stand to benefit [24] on the issue of method-level source code analysis. In most cases, software systems grow by adding new methods and editing of the existing methods [25]. Also, maintenance tasks involve correcting errors generated by faults introduced in the functionality modifications. To understand the complexity of a change, we can investigate the number of unedited methods that rely on the logic of edited methods that implicitly induces a change in the unedited methods. Also, knowledge of edited methods boosts code review activities [26] in the software development industry. Decomposing source code into function-level partitions [27] allows developers to investigate the code at the function level. An Android malware detection tool leverages on method-level correlation relationship of application's API calls makes use of the behavior of the software in various calls in exposing malicious activities in the API [28].

2. METHOD

In this study, three programs were sampled for investigation. The first program was the prototype developed for data gathering through the static analysis of the source code. The prototype mimics a continuous software development scenario where changes are committed on a daily basis. The study also used JSoup version 1.15.4 and version 1.15.3, and Univocity parser version 2.8.4 and version 2.8.3 in the experiments. Both JSoup and Univocity parser programs are open-source products available in github public repository and accessible under Eclipse Public License. JSoup and Univocity parser samples were downloaded on 14th December 2023. These three program samples were chosen to represent practical situations in the software industry where programs are changed for various reasons.

A software system was defined as consisting of a chunk of behavior denoted as a set of methods. Using basic notations, this research defined a program, P as:

- A set of classes forming P: $C – \{c_1, c_2, c_3, ...\}$
- Each class consists of a set of methods $CxM – \{c_x m_1, c_x m_2, c_x m_3, ...\}$
- A set of methods M forming P: $CM – \{CxMn\}$ where x refers to a specific class and n refers to a specific method in the given class.
- Hence the program P was defined as a set of methods: $M – \{m1, m2, m3, ...\}$

When the software behavior changes, the changes are implemented in a method. The attributes of a class may also change. However, a change in the class attributes will be reflected in the methods where the variables are utilized. Following the system definition properties above, a change in the software system can be shown as a change in a set of methods denoted as M'. M' is further defined as M' – $\{C'_x M'_n\}$ where C'x refers to a changed class and M'x refers to a changed method in the class.

The study took two versions of a software source codes as input the modified version of the software and the presumed original version of the software. To perform the method level analysis, the study first cloned the repository containing the earliest program version and the one with the latest edits in a private working space. The study then carried the analysis process to extract edited methods, new methods and the unedited methods associated with the edited methods. The study presents algorithms for extracting target methods from the source code. The algorithms use Java syntax for method definition in extracting the target methods.

2.1. Identification of changed classes and newly added classes

The study generates a checksum for the original and the edited versions of classes. The identification involves a differencing technique. Where the binary checksums for the changed class and the original class differ, the changed class is marked for further analysis. The changed class files and the new class files form the list of candidate classes for further analysis [29]. In cases where no class was edited or added, there is no proceeding with further analysis as this a good conclusion that no system behavior was modified.

2.2. Extraction of changed methods and new methods

From the pool of the new and changed class files, changed methods in each class were identified. The changed methods here include new methods in the edited classes. To keep track of the extracted methods, the algorithm creates method files from both the edited/and new classes and the old classes. A comparison of the two set of method files was made to identify the changed methods and the new methods thereby generating a list of all the edited methods. Also, an extract of all the methods in the new classes was made. The two lists formed one large list of changed methods that was an indication of the possible percentage level of change on the software's behavior implemented in the new system. The extraction of changed methods and new methods can be seen in Algorithms 1 and 2.

Algorithm 1. Extracting changed methods
```
Input:         Input original program version (V) and edited program version (V')
Output:        List of edited methods
Step 1: Create method files for each method in V, V (m)
Step 2: Create method files for each method in V', V' (m)
Step 3: For each method m' in V'(m)
Step 4:        search match m in V(m)
Step 5:         if m' has changed
Step 6:        M' ← Add method m'
Step 7: end for
Step 8: return M'
```

Algorithm 2. Extracting new methods
```
Input:         Input original program version (V) and edited program version (V')
Output:        List of newly added methods
Step 1: get methods in V, V (m)
Step 2: get methods in V', V' (m)
Step 3: For each method m' in V'(m)
Step 4:        if no matching method m in V(m)
Step 6:        $M^{new}$ ← Add method m'
Step 7: end for
Step 8: return $M^{new}$
```

2.3. Extraction of unedited methods associated with changed methods

For each of the edited method, the research established all the methods in the system calling it. These were the associated methods. Some edited methods would be associated with other edited methods or with non-edited methods in the program. The study generated a distinct list of the unedited methods associated with the edited methods, can be seen in Algorithm 3.

Algorithm 3. Extracting unedited methods associated with the changed methods
```
Input:         Input original program version (V) and edited program version (V')
Output:        List of unedited methods calling an edited method
Step 1: get unedited method files in V'
Step 2: get edited method names in V', V' (m)
Step 3: For each method m in V'(m)
Step 4:        for each unedited method file $m^u$ in V' (m)
Step 5:             if m is called in method file $m^u$
Step 6:           $M^u$ ← Add method $m^u$
Step 7:     end for
Step 8: end for
Step 9: return $M^u$
```

3. RESULTS AND DISCUSSION

This study's change impact analysis concentrated on the method-level. Three cases were examined by identifying affected methods. The study presents the results in three cases as shown in Table 1. The study adopts the words changed methods as edited methods + new methods, and affected methods as changed methods + unedited methods associated with edited methods. In Table 2, the changed methods for one of the sampled programs, JSoup, is presented. Table 3 shows, for the sample program JSoup, each edited method with the methods associated to it, both edited and unedited methods.

Table 1. Summarized results of the changes investigation

Item	Prototype	JSoup	UnivocityParser
No. of classes in the edited program	16	59	183
No. of classes in the original program	13	58	183
No. of edited classes	7	19	6
No. of new classes	3	1	0
No. of methods in the program	86	980	1482
No. of edited methods	19	64	49
No. of methods in the new classes	11	13	0
No. of new methods in edited classes	32	7	0
No. of unedited methods associated with edited methods	4	68	37
No. of affected methods	66	152	86

Table 2. JSoup list of edited classes and their changed methods

Class	Edited methods	New methods
CDataNode	text	
Cleaner	clean, isValid	
Collector	collect, head, tail	
Comment	setData	
Consumer		
DataNode	setWholeData	
Document	body, createElement	expectForm, forms
Element	after, appendNormalisedText, appendText, appendWholeText, className, classNames, cssSelector, data, endSourceRange, forEach, getAllElements, getElementsByAttribute, getElementsByAttributeStarting, getElementsByAttributeValue, getElementsByAttributeValueContaining, getElementsByAttributeValueEnding, getElementsByAttributeValueNot, getElementsByAttributeValueStarting, getElementsByClass, getElementsByIndexEquals, getElementsByTag, getElementsContainingText, hasText, head, ownText, parent, parents, prependElement, tagName, tail, text, wholeOwnText,	
HttpConnection	connect, cookies, encodeUrl	
LeafNode		
Node	attr, forEachNode, getDeepChild, hashCode, indent, removeAttr, replaceChild, sourceRange	isNode, normalName
Parser		
QueryParser		
Safelist	isSafeAttribute, removeAttributes	
Selector		
TextNode	clone, lastCharIsWhitespace, splitText	
TokenQueue	consumeCssIdentifier, consumeElementSelector, unescape consumeToIgnoreCase,	consumeEscapedCssIdentifier, escapeCssIdentifier, matchesCssIdentifier
Validate	fail	
XmlDeclaration	getWholeDeclaration	
New classes		
BuildEntities		persist, d, toString, compare, compare, main

Table 3. JSoup sample unedited methods calling edited methods

Edited method	Edited and Unedited methods calling the edited method
text	Document: title, Element: appendElement, Element: children, Element: getAllElements, Element: insertChildren, Element: wholeOwnText, Elements: eachText, Elements: val, Evaluator: matches, HtmlToPlainText: head, ListLinks: main
clean	Cleaner: isValid
isValid	Cleaner: clean
collect	Element: getAllElements, getElementsByAttribute, getElementsByAttributeStarting, getElementsByAttributeValue, getElementsByAttributeValueContaining, getElementsByAttributeValueEnding, getElementsByAttributeValueMatching, getElementsByAttributeValueNot, getElementsByAttributeValueStarting, Element:getElementsByClass, getElementsByIndexEquals, getElementsByIndexGreaterThan, getElementsByIndexLessThan, getElementsByTag, getElementsContainingOwnText, getElementsContainingText, getElementsMatchingOwnText, getElementsMatchingText, Selector: select,
head	Cleaner: isValid, HtmlToPlainText: getPlainText, NodeTraversor: filter, NodeTraversor: traverse
tail	HtmlToPlainText: getPlainText, NodeTraversor: filter, traverse,
consumeElementSelector	QueryParser: byTag
consumeCssIdentifier	QueryParser: byClass, byId, byTag
replaceChild	Node: replaceWith, wrap
sourceRange	Cleaner: createSafeElement, createSafeElement
endSourceRange	Cleaner: createSafeElement
getElementsByClass	Element: getElementsByTag
getElementsByTag	Document: normaliseStructure

Class and method changes represent the overall changes implemented in the system. However, method changes represent operational changes effected in the system. A class change where no method changes were done is considered insignificant with regard to regression testing. Edited and new classes represent a significant component change. Figure 1 is a prototype sample, more details Figure 1(a) shows 30% of the classes were edited and 13% newly-added. Figure 1(b) shows 69% of the methods were changed and 4% of the unchanged methods were affected by the changes made giving a total of 73% of the methods affected by the changes made. The prototype sample presents a case of a software underdevelopment where changes are frequent. Figure 2 explains a JSoup sample, more details Figure 2(a), 33% of edited classes and 2% of new classes show that at least a third of the system was edited. However, the changes with real impact on operational behavior are in the method changes. In this Jsoup sample as shown in Figure 2(b), 16% of the methods were affected by the changes implemented. From 35% of the class changes to 16% of the operational impact shows that several changes concentrated on a few classes. As shown in Table 2, the class “Element” has the largest number of methods edited. Classes like Consumer, LeafNode, Parser, and QueryParser do not have either an edited or a new method. Figure 3 Univocity parser sample, more details Figure 3(a) shows only 3% of the classes were edited and no new class was added. Figure 3(b) shows that of the 1,482 methods in the sample, 3% (49 methods) were edited and 2.5% (37 methods) of the unedited methods were associated to the edited methods totaling to 5.5% of the affected methods.

Our approach largely considered one key software artefact in a program, the method. The list of affected methods ensures that no change escapes the investigation’s test dragnet. Programmers can use the list in navigating the code. Functionality dependencies exposed would also help developers put more emphasis on the code that needs extra attention due to its widespread impact [30]. Where developers and testers are intended to check each method at a time as proposed in [22], [23], our approach presents an improvement in that all the methods of interest are presented to the developers for easy navigation. To expose the indirect effects on the methods that were not edited, this study generates the methods not edited but whose reliance on the results of the edited methods makes them impacted by the changes. This ensures that, after the changes are integrated in whole system, all the possible operational anomalous behavior will be unearthed especially where the changes made in the modified components had adverse effects on the unchanged components or where the changed code sections are utilized in the unchanged components [10].

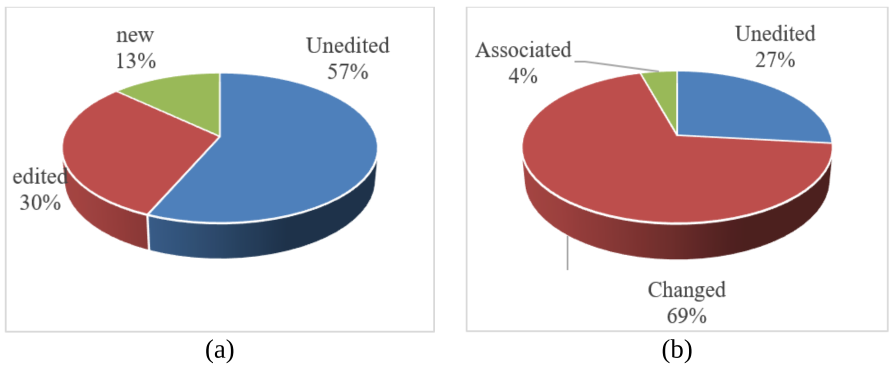

Figure 1. Comparing changes made in the prototype sample classes and changes made in the prototype methods (a) prototype class changes and (b) prototype method changes

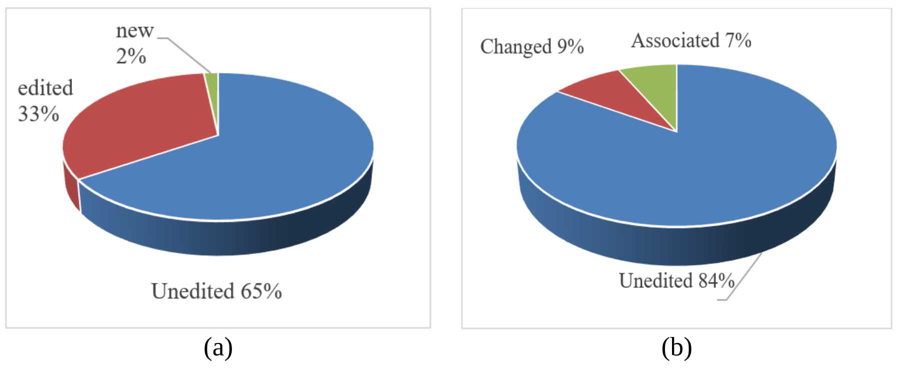

Figure 2. Comparing changes made in the class for JSoup software sample and changes made in the methods (a) JSoup class changes and (b) JSoup method changes

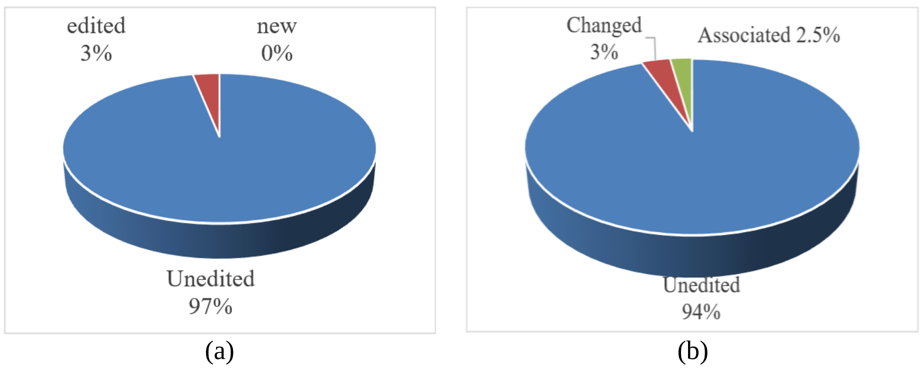

Figure 3. Comparing changes made in the class and changes made in the methods (a) Univocity parser class changes and (b) Univocity parser method changes

4. CONCLUSION

The aim of this study was to investigate the possibility of using method-level static source code analysis in gathering knowledge of the software operational behavior impacted by the changes made in the software system. The research has found that method changes paint a picture of the system behavioral changes that represent the eventual impact of the changes made on the system's operations. Results of the study suggests that changes in a class may not necessarily call for a regression test exercise especially where the changes made do not affect the methods in the class. Since changes affecting the behavior of the system stand at the core of the software regression testing, the research establishes an effective change impact assessment in the overall software under examination. The work presents an improved and simplified approach of analyzing changed source code at the method-level for identifying the changed methods in a java program. The study also, indirectly contributes to the reduction of time and effort that would be spent in understanding change impacts on the changes made in a software operation. In the future work, the study shall evaluate the impact of changes made in software on its system specifications coupled with an investigation on optimization of regression tests selection based on the knowledge of the functionality affected by the changes made in the system.

REFERENCES

[1] E. S. Pasaribu, Y. Asnar, and M. M. I. Liem, "Input injection detection in Java code," in *Proceedings of 2014 International Conference on Data and Software Engineering, ICODSE 2014*, 2014, pp. 1-6, doi: 10.1109/ICODSE.2014.7062698.

[2] E. Söderberg, L. Church, and M. Höst, "Open data-driven usability improvements of static code analysis and its challenges," in *ACM International Conference Proceeding Series*, Jun. 2021, pp. 272–277, doi: 10.1145/3463274.3463808.

[3] V. Lenarduzzi, F. Pecorelli, N. Saarimaki, S. Lujan, and F. Palomba, "A critical comparison on six static analysis tools: detection, agreement, and precision," *Journal of Systems and Software*, vol. 198, p. 111575, 2023, doi: 10.1016/j.jss.2022.111575.

[4] D. Baca, "Identifying security relevant warnings from static code analysis tools through code tainting," in *ARES 2010 - 5th International Conference on Availability, Reliability, and Security*, Feb. 2010, pp. 386–390, doi: 10.1109/ARES.2010.108.

[5] Y. Fang, S. Han, C. Huang, and R. Wu, "TAP: a static analysis model for PHP vulnerabilities based on token and deep learning technology," *PLoS ONE*, vol. 14, no. 11, Nov. 2019, doi: 10.1371/journal.pone.0225196.

[6] S. Arzt, S. Rasthofer, R. Hahn, and E. Bodden, "Using targeted symbolic execution for reducing false-positives in dataflow analysis," in *SOAP 2015 - Proceedings of the 4th ACM SIGPLAN International Workshop on State of the Art in Program Analysis, co-located with PLDI 2015*, 2015, pp. 1–6, doi: 10.1145/2771284.2771285.

[7] L. N. Q. Do, K. Ali, B. Livshits, E. Bodden, J. Smith, and E. Murphy-Hill, "Just-in-time static analysis," *ISSTA 2017 - Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 307–317, 2017, doi: 10.1145/3092703.3092705.

[8] S. Motogna, D. Cristea, D. Șotropa, and A. J. Molnar, "Formal concept analysis model for static code analysis," *Carpathian Journal of Mathematics*, vol. 38, no. 1, pp. 159–168, 2022, doi: 10.37193/CJM.2022.01.13.

[9] B. Guo, Y. W. Kwon, and M. Song, "Decomposing composite changes for code review and regression test selection in evolving software," *Journal of Computer Science and Technology*, vol. 34, no. 2, pp. 416–436, 2019, doi: 10.1007/s11390-019-1917-9.

[10] M. Schnappinger, M. H. Osman, A. Pretschner, and A. Fietzke, "Learning a classifier for prediction of maintainability based on static analysis tools," in *IEEE International Conference on Program Comprehension*, May 2019, vol. 2019-May, pp. 243–248, doi: 10.1109/ICPC.2019.00043.

[11] S. A. Chowdhury, G. Uddin, and R. Holmes, "An empirical study on maintainable method size in Java," in *Proceedings - 2022 Mining Software Repositories Conference, MSR 2022*, 2022, pp. 252–264, doi: 10.1145/3524842.3527975.

[12] P. Emanuelsson and U. Nilsson, "A comparative study of industrial static analysis tools," *Electronic Notes in Theoretical Computer Science*, vol. 217, no. C, pp. 5–21, 2008, doi: 10.1016/j.entcs.2008.06.039.

[13] A. N. Mahmoud, A. Abdelaziz, V. Santos, and M. M. Freire, "A proposed model for detecting defects in software projects," *Indonesian Journal of Electrical Engineering and Computer Science*, vol. 33, no. 1, pp. 290–302, Jan. 2024, doi: 10.11591/ijeecs.v33.i1.pp290-302.

[14] K. A. Mohamed and A. Kamel, "Reverse engineering state and strategy design patterns using static code analysis," *International Journal of Advanced Computer Science and Applications*, vol. 9, no. 1, pp. 568–576, 2018, doi: 10.14569/IJACSA.2018.090178.

[15] T. Azim and I. Neamtiu, "Targeted and depth-first exploration for systematic testing of android apps," *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA*, pp. 641–660, 2013, doi: 10.1145/2509136.2509549.

[16] D. Binkley, D. Lawrie, and C. Morrell, "The need for software specific natural language techniques," *Empirical Software Engineering*, vol. 23, no. 4, pp. 2398–2425, 2018, doi: 10.1007/s10664-017-9566-5.

[17] C. Ragkhitwetsagul, J. Krinke, and D. Clark, "A comparison of code similarity analysers," *Empirical Software Engineering*, vol. 23, no. 4, pp. 2464–2519, 2018, doi: 10.1007/s10664-017-9564-7.

[18] M. Harman and P. O'Hearn, "From start-ups to scale-ups: Opportunities and open problems for static and dynamic program analysis," in *Proceedings - 18th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2018*, Nov. 2018, pp. 1–23, doi: 10.1109/SCAM.2018.00009.

[19] K. Karakaya and E. Bodden, "SootFX: a static code feature extraction tool for Java and Android," in *Proceedings - IEEE 21st International Working Conference on Source Code Analysis and Manipulation, SCAM 2021*, 2021, pp. 181–186, doi: 10.1109/SCAM52516.2021.00030.

[20] A. Brito, A. Hora, and M. T. Valente, "Towards a catalog of composite refactorings," *Journal of Software: Evolution and Process*, Jan. 2023, doi: 10.1002/smr.2530.

[21] A. Hora and R. Robbes, "Characteristics of method extractions in Java: a large scale empirical study," *Empirical Software Engineering*, vol. 25, no. 3, pp. 1798-1833, 2020. doi: 10.1007/s10664-020-09809-8.

[22] F. Grund, S. A. Chowdhury, N. C. Bradley, B. Hall, and R. Holmes, "CodeShovel: constructing method-level source code histories," in *Proceedings - International Conference on Software Engineering*, May 2021, pp. 1510–1522, doi: 10.1109/ICSE43902.2021.00135.

[23] M. Jodavi and N. Tsantalis, "Accurate method and variable tracking in commit history," in *ESEC/FSE 2022 - Proceedings of the 30th ACM Joint Meeting European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, Nov. 2022, pp. 183–195, doi: 10.1145/3540250.3549079.

[24] H. Keuning, B. Heeren, and J. Jeuring, "Code quality issues in student programs," in *Annual Conference on Innovation and Technology in Computer Science Education, ITiCSE*, Jun. 2017, vol. Part F128680, pp. 110–115, doi: 10.1145/3059009.3059061.

[25] D. Steidl and F. Deissenboeck, "How do Java methods grow?," *2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation, SCAM 2015 - Proceedings*, pp. 151–160, 2015, doi: 10.1109/SCAM.2015.7335411.

[26] O. Kononenko, O. Baysal, and M. W. Godfrey, "Code review quality: how developers see it," in *Proceedings - International Conference on Software Engineering*, May 2016, vol. 14-22-May-2016, pp. 1028–1038, doi: 10.1145/2884781.2884840.

[27] Z. Bilgin, M. A. Ersoy, E. U. Soykan, E. Tomur, P. Comak, and L. Karacay, "Vulnerability prediction from source code using machine learning," *IEEE Access*, vol. 8, pp. 150672–150684, 2020, doi: 10.1109/ACCESS.2020.3016774.

[28] H. Zhang, S. Luo, Y. Zhang, and L. Pan, "An efficient android malware detection system based on method-level behavioral semantic analysis," *IEEE Access*, vol. 7, pp. 69246–69256, 2019, doi: 10.1109/ACCESS.2019.2919796.
[29] L. Zhang, "Hybrid regression test selection," in *Proceedings - International Conference on Software Engineering*, 2018, vol. 2018-January, pp. 199–209, doi: 10.1145/3180155.3180198.
[30] S. Jiang, C. McMillan, and R. Santelices, "Do Programmers do Change Impact Analysis in Debugging?," *Empirical Software Engineering*, vol. 22, no. 2, pp. 631–669, 2017, doi: 10.1007/s10664-016-9441-9.

BIOGRAPHIES OF AUTHORS

Fredrick Mugambi Muthengi holds a Master of Science degree in Computer Science from the University of Hull, UK and a Bachelor of Science in Computer Science from Masinde Muliro University of Science and Technology, Kenya. He is Ph.D. student in Computer Science programme at the University of Embu. His research interests are software regression testing, maintaining large software systems and artificial intelligence in education. He can be contacted at email: fremugambi@gmail.com.

Dr. David Muchangi Mugo is a Ph.D. holder of Information Systems from Kenyatta University, Kenya. He has a Master of Science Degree in Computer Science from Technical University of Hamburg, Germany and a Masters of Business Administration where he specialized in Technology Management from Northern Institute of Technology Management, Hamburg, Germany. He graduated with a first-class honour's degree in Bachelor of Science in Computer Science from Kenyatta University. His research interests include ICT for development, electronic health and deployment of artificial intelligence to transform agricultural and health sector. He can be contacted at email: david.mugo@embuni.ac.ke.

Dr. Stephen Makau Mutua holds Ph.D. in Systems Analysis and Integration, a Master in Information Technology and Bachelor of Science in Computer Science. He is an associate professor in the department of Computer Science in Meru University of Science and Technology and currently serving as the dean of the School of Computing and Informatics. He is an established scholar and academician with several publications in refereed journals and book chapters. His research interests are neural networks, computer networks, and data science. He can be contacted at email: smutua@must.ac.ke.

Dr. Faith Mueni Musyoka is a Lecturer in the Department of Computing and Information Technology at the University of Embu, Kenya. She possesses Ph.D. in Information Technology from Kabarak University, Kenya, M.Sc. Information Technology and B.Sc. Computer Science both from Masinde Muliro University of Science and Technology. She is an esteemed member of ACM and OWSD, and has extensive list of publications in well-regarded journals. Her research interests encompass a wide spectrum, from software quality metrics to health informatics. She can be contacted at email: mueni.faith@embuni.ac.ke.