Rochester Institute of Technology
RIT Scholar Works

Theses

5-2022

Leveraging Identifier Naming Structures in Source Code and Bug Reports to Localize Relevant Bugs

James Dugan
jmd9065@rit.edu

Follow this and additional works at: https://scholarworks.rit.edu/theses

Recommended Citation
Dugan, James, "Leveraging Identifier Naming Structures in Source Code and Bug Reports to Localize Relevant Bugs" (2022). Thesis. Rochester Institute of Technology. Accessed from

This Thesis is brought to you for free and open access by RIT Scholar Works. It has been accepted for inclusion in Theses by an authorized administrator of RIT Scholar Works. For more information, please contact ritscholarworks@rit.edu.

Leveraging Identifier Naming Structures in Source Code and Bug Reports to Localize Relevant Bugs

by

James Dugan

A thesis submitted to the
B. Thomas Golisano College of Computing and Information Sciences
Department of Software Engineering
in partial fulfillment of the requirements for the
Master of Science in Software Engineering Degree
at the Rochester Institute of Technology

Approval Commitee:
Dr. Christian D. Newman
Dr. Mohamed Wiem Mkaouer
Dr. J Scott Hawker

May 2022

Abstract

When bugs are found in source code, bug reports are created which contain relevant information for developers to locate and fix the bug. In large source code repositories, it can be difficult and time consuming for developers to manually analyze bug reports to locate a bug. The discovery of patterns between bug reports and source files has led to the creation of automated tools using various techniques. Automated bug localization techniques can reduce the amount of manual effort required by developers by ranking the most probable location of the bug using textual information from bug reports and source code. Although these approaches offer some assistance, the lexical

mismatch between the bug reports and the source code makes it difficult to accurately locate the buggy source code file(s) using Information Retrieval (IR) techniques.

Our research proposes a technique that takes advantage of the lexical and structural patterns observed in source code identifier names to help offset the mismatch between bug reports and their related source code files. Our observations reveal that there are lexical and structural identifier naming trends for different identifier types in the source code. Using two open-source projects, and collecting frequencies for observed identifier patterns across the project, we applied the observed frequencies to matched word occurrences in bug reports across our evaluation data set to modify the significance of that word. Based on observations discovered in our empirical analysis of open source repositories ElasticSearch[1] and RxJava[2], we developed a method to modify the significance of a word by altering the weight of the matched word represented in the Term Frequency - Inverse Document Frequency (TF-IDF) vectorization of that particular bug report. The idea behind this approach is that if we come across a word perceived to be significant based on our observed identifier pattern frequency data, we can apply a weight to that word in the bug report vectorization to increase the cosine similarity score between the bug report and source file vectors.

This work expands and improves upon previous work by Gharibi et al. [1], who propose a multicomponent approach that uses token matching, stack trace, semantic similarity, and a revised vector space model (rVSM). Specifically, our approach modifies the rVSM component, and our work is evaluated

[1]https://github.com/elastic/elasticsearch
[2]https://github.com/ReactiveX/RxJava

on the same three open-source software projects: AspectJ[3], SWT[4], and ZXing[5]. The results of our approach are comparable to the results of Gharibi et al., and we achieve an improvement in some cases. It was observed that our work outperforms many existing bug localization approaches. Top@N, Mean Reciprocal Rank (MRR), and Mean Average Precision (MAP) are metrics used to evaluate and rank our work against other approaches, revealing some improvement in bug localization across three open-source projects.

[3]https://github.com/eclipse/org.aspectj
[4]https://git.eclipse.org/c/platform/eclipse.platform.swt.git/
[5]https://github.com/zxing/zxing

This thesis is dedicated to my family, friends, and Claire for always supporting and being there for me every day.

Acknowledgments

I would like to thank my advisor, Dr. Christian Newman, for his expertise, flexibility, and guidance throughout this research. He has been such a valuable source of information and assistance and it has been a pleasure to work with him. Dr. Newman also motivated me to explore identifier naming patterns in source code. I would also like to thank Dr. Mohamed Wiem Mkaouer for everything I learned in his class and inspiring me to follow through with research in bug localization. Thank you to all the professors who have shared their wisdom with me over the years.

Thank you to my family and friends for encouraging me throughout this work and being a continuous source of love and support.

Contents

1 Introduction **1**

2 Research Objective **5**
2.1 Motivation 5
2.2 Contribution 6
2.3 Research Questions 7

3 Related Work **8**

4 Methodology **13**

5 Analysis Discussion **23**
5.1 RQ1: What identifier naming trends are consistent across multiple projects? 23
5.1.1 Overview of Data 23
5.1.2 Consistent Naming Pattern Observations 29
5.2 RQ2: Does identifier naming structure have an influence on bug localization? 29
5.2.1 Overview of Baseline Approach 29
5.2.2 Our Modified Approach 31
5.2.3 Evaluation and Comparison To Existing Approaches . . 35

6 Threats to Validity **40**
6.1 Internal Validity 40
6.2 External Validity 41
6.3 Construct Validity 42

7 Conclusion & Future Work **43**

List of Figures

4.1 ElasticSearch Bug Report Example 16
4.2 EnrichPolicyRunner.java Removed Lines of Code 16
4.3 TransportDeleteEnrichPolicyAction.java Removed Lines of Code 17
4.4 TransportDeleteEnrichPolicyActionTests.java Removed Lines of Code 18

5.1 RxJava and ElasticSearch Declaration Frequency Plot 26
5.2 RxJava and ElasticSearch Class Frequency Plot 26
5.3 RxJava and ElasticSearch Attribute Frequency Plot 27
5.4 RxJava and ElasticSearch Function Parameter Frequency Plot 27
5.5 RxJava and ElasticSearch Function Name Frequency Plot . . . 28
5.6 RxJava and ElasticSearch Parameter Frequency Plot 28

List of Tables

4.1 Overview of Observed Projects 14
4.2 Report and Source Code Matched Word Count 19
4.3 Matched Word Counts in Relevant Source Files 19
4.4 Identifier Type Counts in Relevant Source Files 20
4.5 Position Breakdown for Matched Identifiers 21
4.6 Part of Speech and Position Breakdown for Matched Identifiers 22

5.1 RxJava Word Position and POS Frequencies 24
5.2 ElasticSearch Word Position and POS Frequencies 24
5.3 Word Position Frequency Difference 24
5.4 Part of Speech Frequency Difference 25
5.5 Overview of Evaluation Datasets 35
5.6 Comparison of the Modified and Baseline rVSM Component . . 36
5.7 Comparison of Overall Performance 38

Chapter 1

Introduction

A software bug is a defect, error, or fault that produces incorrect or unpredictable results [2]. Software bugs are an inevitable byproduct while developing software and can be difficult for developers to resolve in a timely manner in large repositories. Fixing these bugs is one of the most essential activities in software development and maintenance [3]. As repositories evolve and the size of the system increases, the number of bugs in the source code is expected to increase. When bugs are found, developers create bug reports that contain detailed information describing the bug. Developers must then manually sort through the bug reports and numerous files in the source code repository to find the bug's location, wasting crucial time for development. This takes even more time for inexperienced developers, as Zou et al. observed a strong correlation between developer bug reporting behavior and experience and bug fixing rate [3]. The textual information contained in bug reports and source code files can be leveraged by automated techniques to help locate the relevant files that may contain the bug. A significant motivation behind the research and development of automated bug location techniques results from the observation that the cost of fixing a particular bug increases the longer that bug

exists in a software system [4]. Furthermore, many current automated bug localization approaches continue to rank buggy files low in the list of relevant files, which may cause developers to distrust bug localization tools [5].

Many approaches use Information Retrieval (IR) techniques to rank relevant source code files. In these approaches, each bug report acts as a query and the source code files are ranked by their similarity to each report. Developers can then analyze these ranked results to significantly reduce the amount of resources spent on manual bug localization. The effectiveness of these IR approaches is limited by the lexical and structural mismatch between the natural language in the bug reports and the programming language in the source code.

The research that follows in this paper aims to offset the language mismatch and improve bug localization by increasing the significance of matched words (between the source code identifiers and bug reports) using the naming structure frequencies of a given identifier pattern occurrence in a file. This research leverages and improves on previous work by Gharibi et al. [1], who propose a multicomponent approach that uses token matching, stack trace, semantic similarity, and a revised vector space model (rVSM). The motivation behind this work was to determine the relevance of the placement of a word in an identifier and if that placement reveals additional information that can be leveraged to help improve bug localization. Our approach modifies the weight of a word in a Term Frequency Inverse Document Frequency (TF-IDF) vectorization of a bug report by leveraging the frequencies of the observed placement and part of speech tag of the matched word in each identifier type (function name, declaration, class, parameter, function parameter, and attribute).

An initial empirical study is performed to manually observe identifier nam-

ing patterns found between bug reports and their relevant source code files following a grounded theory methodology. Two open-source projects were used for this study: RxJava and ElasticSearch. To gather concrete data to apply to our modified weighing approach, these projects were used to calculate the frequency of matched words appearing in a particular identifier position and part of speech tag for each identifier type. These calculations were collected across both projects and are used to calculate the final weight to apply to a word. For each bug report, based on the frequency values gathered from our matched identifier observation data, we add weight to the matched word in the report if the identifier pattern is frequent. The weight of the word is determined by the type of identifier, the frequency at a particular index, and the frequency of the part of speech tag.

Our approach is evaluated on three open-source software projects: AspectJ, SWT, and ZXing. Top@N, Mean Reciprocal Rank (MRR), and Mean Average Precision (MAP) are metrics used to evaluate our work revealing some improvement across multiple projects and consistent identifier patterns that can be leveraged. When comparing our approach with the rVSM component from Gharibi et al., improvements were observed for ZXing with a Top@1 of 9 (45.0%) over the previous 7 (35.0%). Additionally, for ZXing, we observed improvements of 0.51 over 0.49, and 0.44 over 0.42 for MRR and MAP, respectively. Compared to SWT, our approach achieved a Top@5 of 80 (81.6%) over the previous 79 (80.6%) and consistent results for Top@10. Comparing our approach with AspectJ, we only observed consistent MAP values and no improvements. Combining our approach and the other components proposed by Gharibi et al., we were able to observe an improvement in ZXing with a Top@1 of 13 (65%) over the previous 12 (60%) and an MRR of 0.68 over the

previous 0.67. Consistent results were observed between our approach and the results from Gharibi et al. for SWT and AspectJ using the combined components.

Chapter 2

Research Objective

2.1 Motivation

In software development, on average, as the size of the source code increases, the chance that bugs are introduced into the source code also increases. These bugs become more time consuming and costly to fix the longer the bugs remain in the repository. Therefore, improving automated approaches to rank relevant files is critical to save development time and cost. Developers use identifier names to represent the knowledge recorded in the source code [6]. This enables the ability to observe the consistency of concepts expressed in source code [7] [8]. This consistency in identifier naming can be leveraged to find patterns between bug reports and their corresponding fixed files.

Our approach modifies the significance of a matched word in a bug report based on our identifier naming structure observations and data. Altering the occurrence of a word in a report alters the TF-IDF vector representation of the bug report to bring that bug report and source file closer together in vector space. These vectors are used in a Vector Space Model, specifically a

Revised Vector Space Model (rVSM) originally proposed by Zhou et al. [9] to give larger files additional weight, by increasing the term frequency values, since larger files have been observed to have a higher chance of containing the relevant bug. Although rVSM modifies the vector weight at the file level, we have not found existing work that directly modifies the vector representation of the bug report based on identifier naming structure pattern frequencies at the word level. We aim to answer whether the placement and past of speech of a particular word and the type of the identifier can be used to determine and modify the significance of that word and improve ranking results.

2.2 Contribution

Our primary contribution through this research is to improve bug localization using the observed structural and lexical identifier frequencies. We achieve this by:

1. Empirically and quantitatively identifying source code identifier naming structure patterns for each identifier type across multiple open source projects

2. Building a tool to modify the weight of a term shared between a bug report and source code file based on our observations

3. Improved method of automated bug localization using source code identifier naming structure

2.3 Research Questions

We investigate the naming structure of the source code by performing empirical and quantitative experiments to help answer the following research questions:

- **RQ1:** ***What identifier naming trends are consistent across multiple projects?*** Through our experiments, this question investigates whether there are consistent naming patterns across multiple open-source projects with words shared between bug reports and source code identifiers.
- **RQ2:** ***Does identifier naming structure have an influence on bug localization?*** This question investigates how using the naming patterns of the source code identifiers improves or worsens the ranking results of relevant buggy files.

Chapter 3

Related Work

Several studies have looked at how to improve software maintenance in general [10–89]. Among them, other automated approaches exist to attempt to find a bug's location in software repositories. In this chapter, we present an overview of various bug localization techniques.

Chen et al. [90] developed a method using NLP and a social network model. Since a bug report can be associated with other bug reports through associations with a similar bug, this relationship is added to the network. When an additional bug report is added and found to be related to another report, this relationship generates additional knowledge about the particular bug. The *PageRank* algorithm is then applied to the network of extracted words to rank the fault-prone locations of the software. Although this model was able to generate some insight into where the bug may exist, since this method is only based on *PageRank* to construct the relationships between bug reports, the software sections that have not previously been associated with a bug report are not taken into account in the model.

Lukins et al. [91] developed an LDA-based approach to analyze both the

source code and the bug report. Based on the desired level of granularity (file, class, or method level), each element in the code is preprocessed and stored in a single file used for the LDA model. The source code files are then indexed with the LDA-detected topics. A text query is formed for new bug reports and is searched via a Vector Space Model (VSM). The results using this approach were compared to an approach based on latent semantic indexing (LSI), revealing the LDA technique is at least as accurate as the LSI approach.

In another approach, once a bug report is received, the relevant files associated with the bug can be revealed to the user with *BugLocater*, a bug localization approach developed by Zhou et al. [92]. With *BugLocater*, the software files are ranked based on the textual similarity between the bug report and the source code using a Revised Vector Space Model (rVSM) which considers information about similar bugs that have been previously fixed.

Wang et al. [93] proposed a supervised topic modeling method (STMLocator) to detect relevant source code files given a bug report. This model takes advantage of previous fixes for bugs associated with a bug report. Both the textual and semantic similarities between bug reports and source code files are evaluated. This approach also accounts for bug reports that contain stack traces. This approach achieved a 23.6% improvement for bug localization and 76.2% improvement for bug localization with bug reports containing stack-traces.

Moreno et al. [94] proposed an approach called Lobster (LOcating Bugs using Stack Traces and tExt Retrieval) that combines Text Retrieval (TR) models with static and dynamic analysis. The similarity between the stack traces submitted in the bug reports and their code elements is computed and

combined with the textual similarity. The approach was compared to Lucene, a revised version of VSM, and the results revealed that this approach is at least as effective as Lucene-based localization.

Ye et al. [95] proposed a skip-gram approach to measure the similarities between bug reports and related source code files. This approach connects natural language extracted from bug reports and code snippets as meaning vectors in a shared representation space. To achieve this, word embeddings are trained on API documents, tutorials, and reference documents and then analyzed to evaluate similarities between documents. Word embeddings were found to improve the performance of a simple VSM by connecting natural language and source code.

Swe et al. [95] developed an approach in which class names, functions, and files are processed separately and useful information is extracted from each element. The approach then uses three scores to rank relevant files: stack trace similarity score, bug report similarity score, and class, method, and variable similarity scores.

Sangle et al. [96] propose DRAST, a deep learning and Abstract Syntax Tree (AST) approach that takes into account the syntactic structure of the code. The source code is represented by vectors and combines the use of IR with ML techniques using rVSM and DNNs techniques, respectively.

Zhang et al. [97] proposed an approach that combined code knowledge graph embeddings and a bidirectional attention mechanism. This approach aims to represent the relationships found in the source code and extract the interaction information between bug reports and the source code.

Saha et al. [98] proposed Bug Localization Using Information Retrieval (BLUiR) which breaks a source file into various elements to overcome signif-

icant identifiers being lost in the mix of a large number of identifiers. This approach also considers bug similarity if that data is available for a particular bug. Structured IR with Okapi BM25 scoring is used to compute a final score between the eight different source file elements and those scores are used to rank to source files. It was revealed the results from BLUiR improved upon existing approaches.

Nguyen et al. [99] propose BugScout, an approach that assumes that the textual content of the bug report and the source code share some technical aspects of the system. To represent this relationship, this approach uses a modified LDA topic model to generate topics which are then mapped to the corresponding source code files.

Wong et al. [100] discussed a stack-trace and segmentation analysis technique to achieve better bug location results. Their tool is called BRTracer which is built on top of BugLocator and improves upon BugLocator's results. In this approach, a file a divided into multiple segments with a varying level of granularity. The segment that achieves the highest similarity is then used and the rest of the segments representing the file are ignored. In addition, they proposed a method to identify and give additional weight to files explicitly mentioned in the bug report.

Sisman et al. [101] proposed an approach that incorporates the version history of software files with traditional IR techniques. Specifically, the approach uses information regarding the frequency with which a file is associated with bugs and modification information to determine estimates of the probability that the file contains the source of the bug. It was found by incorporating prior modification history information and a time decay factor in an IR based approach, there was a significant improvement in performance for bug local-

ization.

Rahman et al. [102] propose an approach that extends BugLocator with a Modified Revised Vector Space Model (MrVSM) derived from similar bug reports and source code structure and uses historical data to give certain source code files more weight. It was found that the MrVSM improved upon BugLocator and was able to find a greater number of relevant source code files.

Chapter 4

Methodology

To answer our research questions, we began by making observations following a grounded theory methodology. A grounded theory methodology includes rigorous qualitative data analysis procedures such as open coding, constant comparison, and systematic theory development [103]. Open coding involves collecting potentially useful information without making assumptions about the data to be collected. Constant comparison involves iteratively looking at the collected data and determining whether categories can be formed. Iterative building and refinement of these categories eventually leads to the development of our theory or significant observations. Following this methodology and using the observed data, we discovered patterns and developed a theory on the relevance of identifier naming patterns in bug localization. To develop these observations, we used data from two open-source software projects: RxJava and ElasticSearch. These two projects were used only to build our set of observations and frequency weights and were not used for the evaluation or testing of the approach to eliminate bias and to demonstrate the generalizability of the observations to other existing open-source software projects.

To help gather relevant information to observe, a tool was built to extract previously fixed bug report information. Specifically, the tool extracted the summary and description text from each bug report and the modified lines of code for each modified file associated with that bug report. This tool helped speed up the process of manually observing patterns by gathering all of the information in a single file. Additionally, the output of this tool served as the input data set for the bug localization tool mentioned in the next chapter.

By gathering only the removed lines of code, we can ensure that the observations made are directly related to the bug in the file. In addition, we can ensure that the remaining file changes do not influence our observations since these changes cannot be guaranteed to be related to the particular bug. It should be noted that some bug reports in the repository may have been excluded from our population, since our tool only pulled closed bug reports labeled with a *Bug* tag. Table 4.1 provides a brief overview of the number of closed bug reports in RxJava and ElasticSearch.

Table 4.1: Overview of Observed Projects

Project	Closed Bug Reports	Number of Files
RxJava	264	1977
ElasticSearch	5863	1624

From the 6127 combined bug reports, we used a 95% confidence level and a confidence interval of 10 to determine our total sample size. A confidence level of 10 was selected due to the significant amount of manual effort required in the empirical study. The observation sample size for each project was determined using the following equation:

$$n = \frac{z^2 \cdot p \cdot \frac{1-p}{e^2}}{1 + (z^2 \cdot p \cdot \frac{1-p}{e^2 \cdot N})} \tag{4.1}$$

where:

z = 1.96 (z-value for a 95% confidence level)

p = 0.50 (population proportion)

N = 6127 (population size)

e = 0.10 (confidence level or margin of error)

n = total sample size to observe across RxJava and ElasticSearch

Using Equation 4.1, we determine that a sample size of 95 bug reports will be observed across both projects or 48 bug reports for each individual project. If a confidence level of 5 were used in our sample size calculation, our sample size would increase to 368, which would have significantly limited time allocated to the completion and evaluation of our approach. Observations were made for each bug report while following the iterative process of our methodology. These observations include the number of word occurrences in the report, the number of occurrences in the source file identifiers, the positioning of the word in the identifier, the type of identifier, and the part of speech tag of the matched word in the identifier. An example of a bug report and the observed data is revealed in the following figures. Each word shared between the bug report and the corresponding fixed files is highlighted with a distinct color. These figures are included to show the various placements of matched words in identifiers.

```
Bug Report: 1102122385
Title: Filter enrich policy index deletes to just the policy's
    associated indices.
Summary: If you have two enrich policies, and the name of the
    one plus a dash is a prefix for the name of the other (e.g.
    "policy-test" and "policy-test-more"), then if you delete
    the prefix policy the indices associated with the other
    policy will get deleted, too.
```

Figure 4.1: ElasticSearch Bug Report Example

Each changed file associated with the bug report in Figure 4.1 and the corresponding removed lines of code are represented in Figures 4.2, 4.3 and 4.4.

```
367: String enrichIndexName = EnrichPolicy.getBaseName(
    ↪ policyName) + "-" + nowTimestamp;
```

Figure 4.2: EnrichPolicyRunner.java Removed Lines of Code

```
86: EnrichPolicy policy = EnrichStore.getPolicy(request.getName
    ↪ () state); // ensure the policy exists first
90: throw new ResourceNotFoundException("policy␣[{}]␣not␣found",
    ↪  request.getName());
94: enrichPolicyLocks.lockPolicy(request.getName());
97: List<PipelineConfiguration> pipelines = IngestService.
    ↪ getPipelines(state);
98: List<String> pipelinesWithProcessors = new ArrayList<>();
103: if (processor.getPolicyName().equals(request.getName())) {
113: request.getName()
119: enrichPolicyLocks.releasePolicy(request.getName());
125: GetIndexRequest indices = new GetIndexRequest().indices(
    ↪ EnrichPolicy.getBaseName(request.getName()) + -*)
131: deleteIndicesAndPolicy(concreteIndices, request.getName()
    ↪ ActionListener.wrap((response) -> {
132: enrichPolicyLocks.releasePolicy(request.getName());
141: enrichPolicyLocks.releasePolicy(request.getName());
```

Figure 4.3: TransportDeleteEnrichPolicyAction.java Removed Lines of Code

```
131: createIndex(EnrichPolicy.getBaseName(name) + "-foo1");
132: createIndex(EnrichPolicy.getBaseName(name) + "-foo2");
139: .setindices(EnrichPolicy.getBaseName(name) + "-foo1"
   ↪ EnrichPolicy.getBaseName(name) + "-foo2"),
163: .setindices(EnrichPolicy.getBaseName(name) + "-foo1"
   ↪ EnrichPolicy.getBaseName(name) + "-foo2"),
186: createIndex(EnrichPolicy.getBaseName(name) + "-foo1");
187: createIndex(EnrichPolicy.getBaseName(name) + "-foo2");
56: createIndex(EnrichPolicy.getBaseName(fakeId) + "-foo1");
57: createIndex(EnrichPolicy.getBaseName(fakeId) + "-foo2");
```

Figure 4.4: TransportDeleteEnrichPolicyActionTests.java Removed Lines of Code

The figures above reveal a bug report and the various placements in which a matched word may appear in an identifier. After detecting all the occurrences of the matched words, we collected additional numerical data to obtain a better understanding of the importance in the placement of matched words in source code identifiers.

The total count of each word shared between the bug report and the source code files is represented in Table 4.2.

Table 4.2: Report and Source Code Matched Word Count

Matched Word	Bug Report Count	Source Count
Enrich	2	17
Policy	7	21
Index	2	15
Delete	3	1

Table 4.3 represents each identifier that contained a word mentioned in the bug report. For each matched identifier, the type of identifier and the number of occurrences are recorded.

Table 4.3: Matched Word Counts in Relevant Source Files

Matched Identifiers	Identifier Type	Enrich Count	Policy Count	Index Count	Delete Count
enrichIndexName	Declaration	1	0	1	0
EnrichPolicy	Class	11	11	0	0
enrichPolicyLocks	Attribute	4	4	0	0
EnrichStore	Class	1	0	0	0
policyName	Function Parameter	0	1	0	0
getPolicyName	Function Name	0	1	0	0
releasePolicy	Function Name	0	3	0	0
deleteIndicesAndPolicy	Function Name	0	1	1	1
getPolicy	Function Name	0	1	0	0
lockPolicy	Function Name	0	1	0	0
policy	Declaration	0	1	0	0
getIndexRequest	Function Name	0	0	2	0
indices	Function Name	0	0	1	0
indices	Declaration	0	0	1	0
createIndex	Function Name	0	0	6	0
concreteIndices	Function Parameter	0	0	1	0
setIndices	Function Name	0	0	2	0

We then refined the data in Table 4.3 to obtain the counts for each type

of identifier. Table 4.4 provides an overview of the counts of each type of identifier for each matched word.

Table 4.4: Identifier Type Counts in Relevant Source Files

Identifier Type Counts	Enrich	Policy	Index	Delete
Declaration Count	1	1	2	0
Class Count	12	11	0	0
Attribute Count	4	4	0	0
Function Parameter Count	0	1	1	0
Function Name Count	0	7	12	1
Parameter	0	0	0	0

green!20

Table 4.5 contains the matched word position data for each identifier type

Table 4.5: Position Breakdown for Matched Identifiers

Identifier Type	Matched Word Position in Identifier	Occurrence Count
Declaration	1 of 1	1
	1 of 3	2
	2 of 3	3
Class	1 of 2	12
	2 of 2	11
Attribute	1 of 3	4
	2 of 3	4
Function Parameter	1 of 2	1
	2 of 2	1
Function Name	1 of 1	1
	1 of 4	1
	2 of 2	13
	2 of 3	3
	3 of 4	1
	4 of 4	1

Part of speech information is also collected for each identifier type with the associated position of the matched word. Including the part of speech tag of the word enables us to have a better idea of where a given word in the bug report is more likely to appear in the identifier name and it serves as another feature we can leverage to determine the significance of matched identifier words. Table 4.6 contains the matched word part of speech and position data for each identifier type. The number in parentheses represents the number of times a matched word was in that particular identifier position.

Table 4.6: Part of Speech and Position Breakdown for Matched Identifiers

Identifier Type	Noun Modifier	Noun	Verb	Conjunction
Declaration	1 of 3 (1) 2 of 3 (1)	1 of 1 (1)		
Class	1 of 2 (11) 1 of 2 (1)	2 of 2 (11)		
Attribute	1 of 3 (4) 2 of 3 (4)			
Function Parameter	1 of 2 (1)	2 of 2 (1)		
Function Name	2 of 3 (3) 2 of 4 (1)	1 of 1 (1) 2 of 2 (13) 4 of 4 (1)	1 of 4 (1)	3 of 4 (1)

The information in this table was collected for each bug report in an iterative process as the significant features to collect became apparent. Table 4.6 contains the most useful data we can leverage to develop our approach and data collection tool.

green!20

Chapter 5

Analysis Discussion

In this chapter, the answers to our research questions are presented by discussing the observations of identifier naming patterns and the effectiveness of leveraging the identifier naming patterns in bug localization.

5.1 RQ1: What identifier naming trends are consistent across multiple projects?

To answer this research question, we break this section into two parts: an overview of the data generated from our tool and a discussion of our identifier naming pattern observations from the empirical study and generated data.

5.1.1 Overview of Data

This section gives an overview of the identifier naming data from RxJava and ElasticSearch that is used to derive our concrete observations.

Tables 5.1 and 5.2 reveal the position, noun, and verb frequencies for each position index and identifier type in RxJava and ElasticSearch, respectively.

Table 5.1: RxJava Word Position and POS Frequencies

Identifier Type	First Index Frequency	Second Index Frequency	Second Last Index Frequency	Last Index Frequency	MID Frequency	Noun Frequency	Verb Frequency
Declaration	.720	.062	.019	.198	0.00	.856	.103
Class	.287	.344	.117	.226	.023	.707	.056
Attribute	.837	0.00	0.00	.162	0.00	.864	.135
Function Parameter	.491	.062	.070	.375	0.00	.860	.096
Function Name	.359	.222	.068	.226	.122	.800	.082
Parameter	.607	.016	.009	.366	0.00	.770	.094

Table 5.2: ElasticSearch Word Position and POS Frequencies

Identifier Type	First Index Frequency	Second Index Frequency	Second Last Index Frequency	Last Index Frequency	MID Frequency	Noun Frequency	Verb Frequency
Declaration	.650	.088	.035	.224	0.00	.804	.059
Class	.311	.196	.131	.245	.114	.918	.065
Attribute	.660	0.00	0.00	.301	0.00	.867	0.00
Function Parameter	.747	.020	.010	.222	0.00	.797	.010
Function Name	.503	.171	.036	.226	.061	.773	.202
Parameter	.789	.026	.008	.175	0.00	.824	.008

Table 5.3 represents the absolute difference between the position frequency values found in Table 5.1 and Table 5.2.

Table 5.3: Word Position Frequency Difference

Identifier Type	First Index Frequency Difference	Second Index Frequency Difference	Second Last Index Frequency Difference	Last Index Frequency Difference	MID Frequency Difference	Total Frequency Difference
Declaration	.070	.026	.016	.026	0.00	.138
Class	.024	.148	.014	.019	.091	.296
Attribute	.177	0.00	0.00	.139	0.00	.316
Function Parameter	.256	.042	.060	.153	0.00	.511
Function Name	.144	.051	.032	0.00	.061	.288
Parameter	.182	.010	.001	.191	0.00	.384

Table 5.4 represents the absolute difference between the noun and verb part of speech frequencies found in Table 5.1 and Table 5.2.

Table 5.4: Part of Speech Frequency Difference

Identifier Type	Noun Frequency Difference	Verb Frequency Difference	Frequency Difference
Declaration	.052	.044	.096
Class	.211	.009	.220
Attribute	.003	.135	.138
Function Parameter	.063	.086	.149
Function Name	.027	.120	.147
Parameter	.054	.086	.140

The following plots represent the position data in Table 5.1 and Table 5.2 for each identifier type.

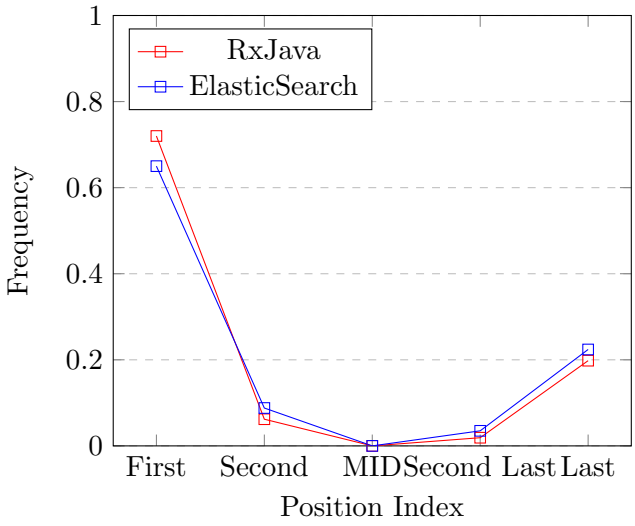

Figure 5.1: RxJava and ElasticSearch Declaration Frequency Plot

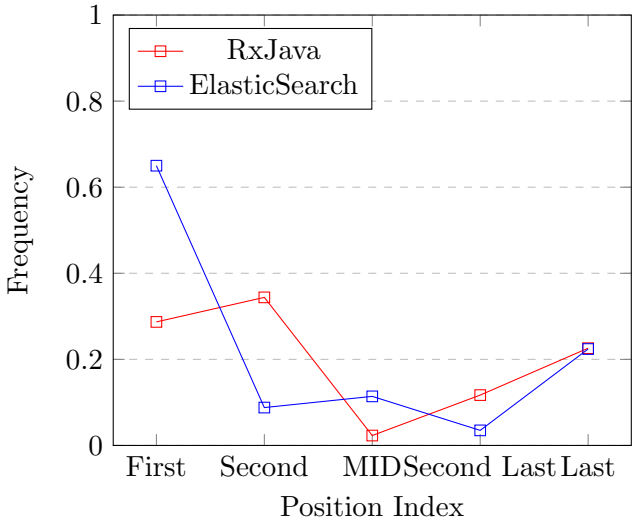

Figure 5.2: RxJava and ElasticSearch Class Frequency Plot

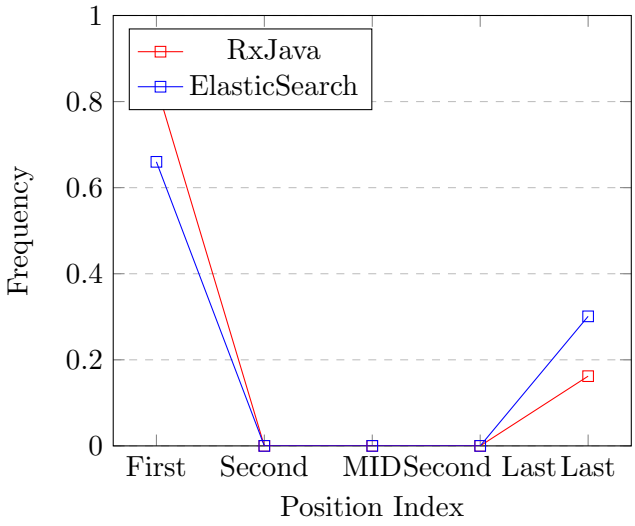

Figure 5.3: RxJava and ElasticSearch Attribute Frequency Plot

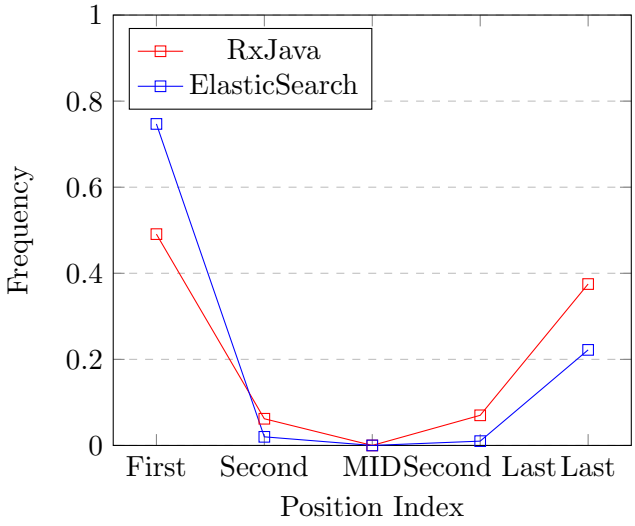

Figure 5.4: RxJava and ElasticSearch Function Parameter Frequency Plot

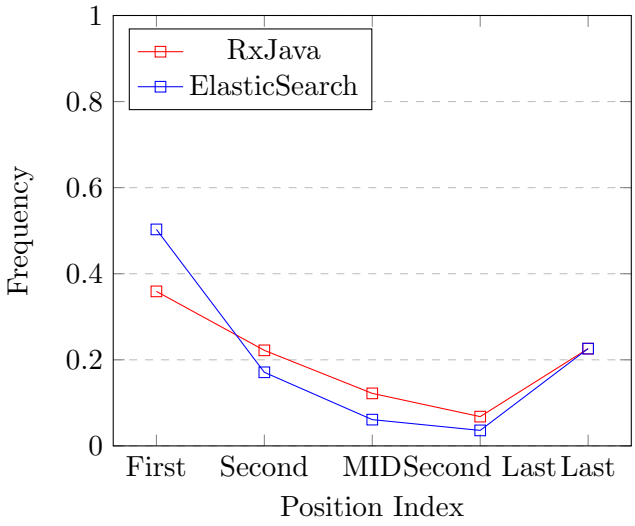

Figure 5.5: RxJava and ElasticSearch Function Name Frequency Plot

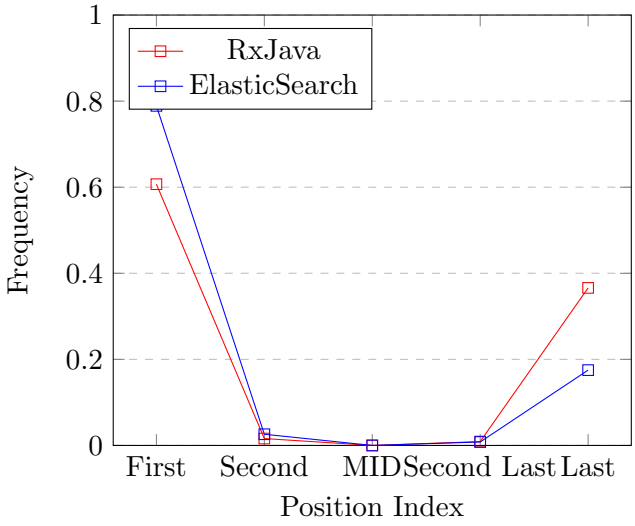

Figure 5.6: RxJava and ElasticSearch Parameter Frequency Plot

5.1.2 Consistent Naming Pattern Observations

Based on our empirical study and the data obtained by our selection of significant features represented in the previous section, we established the following high-level observations:

1. The significance of a particular word decreases as its position moves towards the middle of the identifier
2. Matched words found in declaration identifiers have the most consistent positioning and part of speech tag
3. The variance of the frequency for a particular identifier type increases as the the position of a word moves towards the ends of an identifier
4. The frequency of the last index position in function name identifiers is consistent

5.2 RQ2: Does identifier naming structure have an influence on bug localization?

To answer this research question, we will break this section into three subsections: an overview of the baseline approach proposed by Gharibi et al. [1], a discussion of our modified rVSM approach, and a comparison between our approach and other bug localization tools using the same data set.

5.2.1 Overview of Baseline Approach

Our research improves a multicomponent tool that uses a combination of token matching, stack trace analysis, semantic similarity, and rVSM to calculate

similarity scores between each bug report and source code file. These scores are then combined into a single result and the weight of each individual score from each component is optimized by an objective function. Although our approach specifically modifies only the rVSM component of the baseline tool, we use the other components in the approach in our evaluation to compare our results with Gharibi et al. and other bug localization approaches.

In the baseline tool, the comments and identifier names in the source code are preprocesed by tokenizing, stemming, and removing noise in the text including stop words, spaces, Java keywords, and punctuation. In addition to the source files, the title and summary of the bug reports are parsed and preprocessed using the same preprocessing techniques used on the source files. The part of speech tag for each token in a bug report and source file identifier is also processed and included in the data. The token matching component attempts to assist bug localization by finding exact token matches between the bug report and the source code. If a full token match is not found, the component can also find partial token matches, which results in a lower score between the report and source file compared to a full match. Schroter et al. reveal that about 90% of bugs are located in the top ten stack trace frames [104]. It was also observed that most of the buggy files are explicitly mentioned in the stack trace information. These observations motivated the inclusion of a stack trace component in this the baseline approach. The stack trace component leverages bug reports that contain stack trace information and files mentioned directly in the stack trace improve results. The semantic similarity component aims to reduce the lexical mismatch between the bug reports and the relevant source code files. This component specifically uses Word2Vec [105], an algorithm that learns the meaning behind words, and GloVe [106], an al-

gorithm that constructs word vectors. Using pretrained word vectors to build vectors of the bug reports and source files, the cosine similarity is calculated between each bug report and source code file representation. In the rVSM similarity component, each bug report and source code file are represented as a vector of term weights. The summary and descriptions from bug reports are used to represent bug reports, and source files are represented by select identifier names and comments. Each bug report and source file is then vectorized using TF-IDF to represent the bug reports and source files as vectors of term weights. The cosine similarity is calculated between each bug report and source file and weights are applied to the similarity scores to give larger files a higher score.

5.2.2 Our Modified Approach

Preprocessing

Compared to the approach of Gharibi et al., our modified rVSM component does not preprocesses comments in the source code, only identifiers in the source code. The title and summary text from bug reports are used to represent each bug report. To extract the relevant identifier information in each source code file, we use a tool that uses srcML [107], a tool that creates an XML representation of a source file. After retrieving the identifiers, we collect the name and type of each identifier. Each identifier is then split into individual terms, which together represent the full identifier name. Spiral [108], an identifier name splitting tool, was used to help split identifiers by camel case, underscores, words, numbers, etc. Each identifier token is then stemmed using the same porter stemmer used in the baseline approach.

Vectorization

Now that the bug report and the source code identifier strings are prepossessed, we vectorize each report and file using TF-IDF. Breaking down TF-IDF into its two respective parts, TF (Term Frequency) represents the number of times a term occurs in a particular file, while IDF (Inverse-Document Frequency) represents the frequency of a word across the set of files. The weights of the terms in these vectors represent the significance of a particular term in a particular file across the entire set of files.

TF-IDF is defined below in Equation 5.1:

$$w_{i,j} = tf_{i,j} \times log(\frac{N}{df_i}) \tag{5.1}$$

where:

$tf_{i,j}$ = number of occurrences of term i in file j
$df_{i,j}$ = number of occurrences containing i
N = total number of files

Obtaining Weights

Using the average of the frequency data collected with our helper tool for both observation projects, we can obtain the expected frequency of the occurrence of a particular identifier pattern. For example, if a matched word is found to be in the second index position of an identifier and the matched word is a noun, we can retrieve the average occurrence frequency of this particular pattern. These frequencies are significant because they help increase or decrease the weight of a matched word observed in a bug report based on our previously collected frequency data. For each bug report, our approach iterates through

each source file. The goal is to build a map of matched words between each bug report and all source code files and weights representing the normalized sum of frequencies collected based on each identifier pattern observed containing that matched word.

After building this set of term weights for each bug report, we modify the original TF-IDF vectorization of each bug report. This modification alters the significance of matched words in a bug report based on the frequency of that particular pattern from our observation data. This is achieved by normalizing the weights and centering the weights around a value of 1, so if there are no observations of matched words or if the frequency of a particular pattern is 0, our modification will not change the original TF-IDF term weights. Our modification weights for each term in a bug report align with each original TF-IDF term weight, so we iterate through each bug report and multiply the modification vectors with the original TF-IDF vectors, resulting in the final bug report term weights.

These vectors are then normalized by the Euclidean norm defined in Equation 5.2:

$$V_{norm} = \frac{\vec{V}}{\sqrt{w_1^2 + w_2^2 + ... + w_v^2}} \tag{5.2}$$

where:

V = a vector of term weights

w = term weight in vector V

Similarity

To determine the similarity between each bug report and each source file, we calculate the cosine similarity between each pair of vectors.

Cosine similarity is defined below in Equation 5.3:

$$cos(r, c) = \frac{\vec{V_r} \cdot \vec{V_c}}{|\vec{V_r}| \times |\vec{V_s}|} \tag{5.3}$$

where:

r = bug report

c = source code file

$\vec{V_r}$ = vector representation of r

$\vec{V_c}$ = vector representation of c

Similar to the baseline approach, we apply additional weight to larger files since it has been observed larger files are more likely to contain the bug. To determine the additional weight to apply to each source file, we use Equations 5.4 and 5.5 below:

Equation 5.4 represents the normalization function for the source file scores for each bug report.

$$nor(x) = \frac{x - x_{min}}{x_{max} - x_{min}} \tag{5.4}$$

where:

x = set of original scores

x_{min} = minimum value in x

x_{max} = maximum value in x

Using this normalization function, we compute the weights to apply to each similarity score using Equation 5.5 below:

$$L_{s,x} = \frac{1}{1 + e^{-\lambda \times nor(x)}} \tag{5.5}$$

where:

s = source file

x = number of terms in s

L = length score of s

λ = weight to adjust the impact of L on the original similarities

Using Equations 5.3 and 5.5, we calculate the final similarity score between each bug report and source file defined in Equation 5.7 below:

$$Score = cos(r, c) \times L_{s,x} \tag{5.6}$$

5.2.3 Evaluation and Comparison To Existing Approaches

Our approach was evaluated using three open-source projects: AspectJ, SWT, and ZXing. An overview of these projects can be found in Table 5.5.

Table 5.5: Overview of Evaluation Datasets

Project	Study Period	Number of Fixed Bugs	Number of Source Files
AspectJ	Jul 2002 - May 2010	286	6485
SWT	Oct 2004 - Apr 2010	98	484
ZXing	Mar 2010 - Sep 2010	20	391

Top@N, Mean Average Precision (MAP), and Mean Reciprocal Rank (MRR) will be used as our evaluation metrics to determine the overall effectiveness of the proposed approach and compare different variables within the approach to other bug localization tools.

The following is a brief overview of the representation and purpose of each metric:

- Top@N measures the accuracy of at least one source file containing a bug appearing in the results for the top N results, where N is 1, 5, and 10 for this proposed research
- MAP is a standard metric used in IR techniques as it considers both the accuracy and ranking result. This metric is the average performance of locating all relevant files.
- MRR is another standard metric used in IR techniques and it measures the relation between bug reports and predicted source files by finding the position of first bug correctly located in the predictions. As MRR increases, bug localization performance is improving.

Before showing the overall results of our approach, which uses a combination of each component mentioned previously, the table below compares the similarity evaluation results of our rVSM component modified by identifier naming structures and the rVSM component proposed by Gharibi et al.

Table 5.6: Comparison of the Modified and Baseline rVSM Component

Project	Approach	Top@1	Top@5	Top@10	MRR	MAP
AspectJ	Our approach	88 (30.7%)	148 (51.7%)	**179 (62.6%)**	0.41	**0.23**
	Gharibi et al.	**90 (31.5%)**	**150 (52.4%)**	**179 (62.6%)**	**0.42**	**0.23**
SWT	Our approach	39 (39.8%)	**80 (81.6%)**	**85 (86.7%)**	0.57	0.50
	Gharibi et al.	**42 (42.9%)**	79 (71.4%)	**85 (86.7%)**	**0.59**	**0.51**
ZXing	Our approach	**9 (45.0%)**	11 (70.0%)	12 (60.0%)	**0.51**	**0.44**
	Gharibi et al.	7 (35.0%)	**14 (70.0%)**	**15 (75.0%)**	0.49	0.42

Table 5.6 reveals that our approach achieved better results with smaller projects. Considering the evaluation results from ZXing and SWT, which

have 391 and 484 source files, our modified VSM approach was able to achieve better performance in the Top@1 for ZXing and Top@5 for SWT compared to Gharibi et al. Additionally, we were able to achieve better MRR and MAP for ZXing with 0.51 and 0.44, respectively. Compared to the baseline rVSM component, where ZXing achieved a 0.49 MRR and a 0.42 MAP. Compared to these two projects, AspectJ is significantly larger in size with 6485 files, about 15 times the size of the two smaller projects. When observing the difference in size, it appears that our performance decreased as the size of the project increased. Our evaluation results for AspectJ compared to the other results reveal that there is a reduction in performance. Since we had a smaller sample of data, some identifier patterns for a particular index never appeared, so they were ignored in our weighing approach. If we gather our frequency data on a larger project to use in our approach, the frequency of these patterns could be greater than 0 and now alter the significance of that pattern occurrence in a source file.

The overall results of our approach are compared with the results of other bug localization tools in Table 5.7:

Table 5.7: Comparison of Overall Performance

Project	Approach	Top@1	Top@5	Top@10	MRR	MAP
AspectJ	Our approach	**133 (46.5%)**	**192 (67.1%)**	**212 (74.1%)**	**0.56**	**0.32**
	Gharibi et al.	**133 (46.5%)**	**192 (67.1%)**	**212 (74.1%)**	**0.56**	**0.32**
	AmaLgam	127 (44.4%)	187 (65.4%)	209 (73.1%)	0.54	0.33
	Rahman et al.	N/A	N/A	N/A	N/A	N/A
	BRTracer	113 (39.5%)	173 (60.5%)	197 (68.9%)	0.49	0.29
	BLUiR	97 (33.9%)	150 (52.4%)	176 (61.5%)	0.43	0.25
	BugLocator	88 (30.8%)	146 (51.0%)	170 (59.4%)	0.41	0.22
SWT	Our approach	**67 (68.4%)**	**84 (85.7%)**	**88 (89.8%)**	**0.76**	0.64
	Gharibi et al.	**67 (68.4%)**	**84 (85.7%)**	**88 (89.8%)**	**0.76**	**0.65**
	AmaLgam	61 (62.2%)	80 (81.6%)	**88 (89.8%)**	0.71	0.62
	Rahman et al.	47 (48.0%)	70 (71.4%)	81 (82.7%)	0.60	0.54
	BRTracer	46 (46.9%)	78 (79.6%)	87 (88.8%)	0.59	0.53
	BLUiR	55 (56.1%)	75 (76.5%)	86 (87.8%)	0.66	0.58
	BugLocator	39 (39.8%)	66 (67.3%)	80 (81.6%)	0.53	0.45
ZXing	Our approach	**13 (65.0%)**	14 (70.0%)	**16 (80.0%)**	**0.68**	0.53
	Gharibi et al.	12 (60.0%)	**15 (75.0%)**	**16 (80.0%)**	0.67	**0.56**
	AmaLgam	8 (40.0%)	13 (65.0%)	14 (70.0%)	0.51	0.41
	Rahman et al.	9 (45.0%)	14 (70.0%)	15 (75.0%)	0.55	0.50
	BRTracer	10 (50.0%)	13 (65.0%)	15 (75.0%)	0.61	0.49
	BLUiR	8 (40.0%)	13 (65.0%)	14 (70.0%)	0.49	0.39
	BugLocator	8 (40.0%)	12 (60.0%)	14 (70.0%)	0.50	0.44

In addition to class and file names, Gharibi et al. use source code comments to help resolve the lexical mismatch between the bug reports and source code files. In our approach, we wanted to explicitly focus on leveraging the naming structure of identifiers, so all comments in the files were ignored. Overall, based on the evaluation data, our approach was able to still perform compar-

atively to the results of Gharibi et al. and better than AmaLgam, Rahman et al., BRTracer, BLUiR, and BugLocator

ElasticSearch and RxJava, the projects used for our observations, had 1624 and 1977 files, respectively. These projects are about 4 times the size of ZXing and SWT. There is a chance that our improvements are limited by our lack of diversity in the number of files for the projects we observed. We ensured that the projects varied in terms of number of bug reports (5863 for ElasticSearch and 264 for RxJava), but the number of files for a particular project should be considered for future work to improve results. Additional observations must be made in larger projects to capture the full breadth of identifier patterns in the source code.

Chapter 6

Threats to Validity

This chapter presents threats that may impact the generalizability and applicability of our observations to other software systems. These threats are broken down into internal, external, and construct validity. Internal validity refers to experimental errors, external validity refers to the generalizability of our results, and construct validity refers to the correctness of the metrics used in the experimental evaluation.

6.1 Internal Validity

A threat to internal validity is the relatively small number of bug reports observed in the empirical analysis. The small sample size limited the number of observations we were able to construct and it is also significant to consider there is a 10% chance the observations from our empirical study are not repeatable and may contain a bias.

To reduce internal validity, we evaluated and compared our approach with other approaches using the same dataset as several previous approaches [1]

[102] [98] [100] [9]. Additionally, the same evaluation metrics were used for these comparisons. We also derived our observations and tested our tool on two projects that were not used for evaluation. This was done to eliminate bias in our technique and increase the generalizability of our approach, since we are not overfitting any particular project. It is possible the two projects we used for our observations and testing do not accurately represent the nature of the other three projects used in our evaluation. Since our weighting technique relied heavily on the weights generated using these two projects, it is likely our approach could be improved by collecting observation data on a larger sample of projects.

6.2 External Validity

Since our approach is highly dependent on the proper naming of identifiers, identifiers that do not follow a standard writing style pose a threat to the success of our approach. To help mitigate this threat, we used three open source projects of varying sizes that were developed independently [1]. Another external threat to consider is the programming language of choice analyzed for this research. For this work, all of the observed source code is written in Java. The identifier naming structure observations will likely not completely generalize to projects written in other languages. However, future work could expand on the observations mentioned previously to apply this approach to other languages.

6.3 Construct Validity

We mitigated threats to construct validity using Top@N, MRR, and MAP to evaluate our approach. These are standard metrics that are used to demonstrate improvement with our approach compared to other previous approaches.

Chapter 7

Conclusion & Future Work

Locating bugs in source code is a time consuming and costly task for developers. Automated bug localization approaches help to find buggy files by ranking relevant files associated with each bug report to reduce the number of files developers must manually sort through.

The objective of this work was to improve bug localization using observed patterns and concrete data from matched words in bug reports and their relevant source code identifier names. To achieve this, we have performed an empirical study on two open-source projects to derive the features we want to capture and build towards a collection of consistent patterns between words found in a bug report and in source code identifiers. Using the features observed in the empirical study, a tool was created to gather relevant concrete data for each determined feature. Using the concrete data, observations were made consistent with various data from both RxJava and ElasticSearch. Our approach shows improvement in bug localization in particular cases and shows that there are consistent patterns in identifier naming structure which can be leveraged to improve bug localization.

This research can be improved by incorporating a genetic algorithm to fine-tune hyperparameters to optimize the weights applied to the occurrence of a matched word to improve results. Furthermore, by observing a larger sample, additional patterns can be collected to further improve the results of the approach.

Bibliography

[1] Reza Gharibi, Amir Hossein Rasekh, Mohammad Hadi Sadreddini, and Seyed Mostafa Fakhrahmad. Leveraging textual properties of bug reports to localize relevant source files. *Information Processing Management*, pages 1058–1076, 2018.

[2] Faiza Khan, Summrina Kanwal, Sultan Alamri, and Bushra Mumtaz. Hyper-parameter optimization of classifiers, using an artificial immune network and its application to software bug prediction. *IEEE Access*, pages 20954–20964, 2020.

[3] Weiqin Zou, Xin Xia, Weiqiang Zhang, Zhenyu Chen, and David Lo. An empirical study of bug fixing rate. *2015 IEEE 39th Annual Computer Software and Applications Conference*, 2:254–263, 2015.

[4] Marcelo Garnier, Isabella Ferreira, and Alessandro Garcia. On the influence of program constructs on bug localization effectiveness. *Journal of Software Engineering Research and Development*, page 6, 2017.

[5] Tien-Duy B. Le, Ferdian Thung, and David Lo. Predicting effectiveness of ir-based bug localization techniques. In *2014 IEEE 25th International Symposium on Software Reliability Engineering*, pages 335–345, 2014.

[6] Simon Butler. Mining java class identifier naming conventions. *2012 34th International Conference on Software Engineering (ICSE)*, pages 1641–1643, 2012.

[7] J.-R. Falleri, M. Huchard, M. Lafourcade, C. Nebut, V. Prince, and M. Dao. Automatic extraction of a wordnet-like identifier network from software. pages 4–13, 2010.

[8] Jan Nonnen, Daniel Speicher, and Paul Imhoff. Locating the meaning of terms in source code research on "term introduction". *2011 18th Working Conference on Reverse Engineering*, pages 99–108, 2011.

[9] Jian Zhou, Hongyu Zhang, and David Lo. Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports. *2012 34th International Conference on Software Engineering (ICSE)*, pages 14–24, 2012.

[10] Christian D Newman, Michael J Decker, Reem Alsuhaibani, Anthony Peruma, Mohamed Mkaouer, Satyajit Mohapatra, Tejal Vishoi, Marcos Zampieri, Timothy Sheldon, and Emily Hill. An ensemble approach for annotating source code identifiers with part-of-speech tags. *IEEE Transactions on Software Engineering*, 2021.

[11] Yaroslav Golubev, Zarina Kurbatova, Eman Abdullah AlOmar, Timofey Bryksin, and Mohamed Wiem Mkaouer. One thousand and one stories: a large-scale survey of software refactoring. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1303–1313, 2021.

[12] Eman Abdullah AlOmar, Anthony Peruma, Mohamed Wiem Mkaouer, Christian D Newman, and Ali Ouni. Behind the scenes: On the relationship between developer experience and refactoring. *Journal of Software: Evolution and Process*, page e2395, 2021.

[13] Eman Abdullah AlOmar, Mohamed Wiem Mkaouer, Christian Newman, and Ali Ouni. On preserving the behavior in software refactoring: A systematic mapping study. *Information and Software Technology*, 140:106675, 2021.

[14] Eman Abdullah AlOmar, Ben Christians, Mihal Busho, Ahmed Hamad AlKhalid, Ali Ouni, Christian Newman, and Mohamed Wiem Mkaouer. Satdbailiff-mining and tracking self-admitted technical debt. *Science of Computer Programming*, 213:102693, 2022.

[15] Anthony Peruma, Steven Simmons, Eman Abdullah AlOmar, Christian D Newman, Mohamed Wiem Mkaouer, and Ali Ouni. How do i refactor this? an empirical study on refactoring trends and topics in stack overflow. *Empirical Software Engineering*, 27(1):1–43, 2022.

[16] Eman Abdullah AlOmar, Jiaqian Liu, Kenneth Addo, Mohamed Wiem Mkaouer, Christian Newman, Ali Ouni, and Zhe Yu. On the documentation of refactoring types. *Automated Software Engineering*, 29(1):1–40, 2022.

[17] Eman Abdullah Alomar, Tianjia Wang, Vaibhavi Raut, Mohamed Wiem Mkaouer, Christian Newman, and Ali Ouni. Refactoring for reuse: an empirical study. *Innovations in Systems and Software Engineering*, pages 1–31, 2022.

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

[19] Eman Abdullah AlOmar, Mohamed Wiem Mkaouer, and Ali Ouni. Mining and managing big data refactoring for design improvement: Are we there yet? *Knowledge Management in the Development of Data-Intensive Systems*, pages 127–140, 2021.

[20] Wajdi Aljedaani, Anthony Peruma, Ahmed Aljohani, Mazen Alotaibi, Mohamed Wiem Mkaouer, Ali Ouni, Christian D Newman, Abdullatif Ghallab, and Stephanie Ludi. Test smell detection tools: A systematic mapping study. *Evaluation and Assessment in Software Engineering*, pages 170–180, 2021.

[21] Hussein Alrubaye, Mohamed Wiem Mkaouer, and Ali Ouni. On the use of information retrieval to automate the detection of third-party java library migration at the method level. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, pages 347–357. IEEE, 2019.

[22] Hussein Alrubaye, Mohamed Wiem Mkaouer, Igor Khokhlov, Leon Reznik, Ali Ouni, and Jason Mcgoff. Learning to recommend third-party library migration opportunities at the api level. *Applied Soft Computing*, 90:106140, 2020.

[23] Hussein Alrubaye, Stephanie Ludi, and Mohamed Wiem Mkaouer. Comparison of block-based and hybrid-based environments in trans-

ferring programming skills to text-based environments. *arXiv preprint arXiv:1906.03060*, 2019.

[24] Anthony Peruma, Mohamed Wiem Mkaouer, Michael John Decker, and Christian Donald Newman. Contextualizing rename decisions using refactorings and commit messages. In *2019 19th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 74–85. IEEE, 2019.

[25] Christian D Newman, Mohamed Wiem Mkaouer, Michael L Collard, and Jonathan I Maletic. A study on developer perception of transformation languages for refactoring. In *Proceedings of the 2nd International Workshop on Refactoring*, pages 34–41, 2018.

[26] Hussein Alrubaye and Mohamed Wiem Mkaouer. Automating the detection of third-party java library migration at the function level. In *CASCON*, pages 60–71, 2018.

[27] Hussein Alrubaye, Mohamed Wiem Mkaouer, and Anthony Peruma. Variability in library evolution: An exploratory study on open-source java libraries. In *Software Engineering for Variability Intensive Systems*, pages 295–320. Auerbach Publications, 2019.

[28] Montassar Ben Messaoud, Ilyes Jenhani, Nermine Ben Jemaa, and Mohamed Wiem Mkaouer. A multi-label active learning approach for mobile app user review classification. In *International Conference on Knowledge Science, Engineering and Management*, pages 805–816. Springer, 2019.

[29] Hussein Alrubaye, Mohamed Wiem Mkaouer, and Ali Ouni. Migrationminer: An automated detection tool of third-party java library migration at the method level. In *2019 IEEE international conference on software maintenance and evolution (ICSME)*, pages 414–417. IEEE, 2019.

[30] Deema Alshoaibi, Kevin Hannigan, Hiten Gupta, and Mohamed Wiem Mkaouer. Price: Detection of performance regression introducing code changes using static and dynamic metrics. In *International Symposium on Search Based Software Engineering*, pages 75–88. Springer, 2019.

[31] Eman Abdullah AlOmar, Mohamed Wiem Mkaouer, Ali Ouni, and Marouane Kessentini. On the impact of refactoring on the relationship between quality attributes and design metrics. In *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–11. IEEE, 2019.

[32] Licelot Marmolejos, Eman Abdullah AlOmar, Mohamed Wiem Mkaouer, Christian Newman, and Ali Ouni. On the use of textual feature extraction techniques to support the automated detection of refactoring documentation. *Innovations in Systems and Software Engineering*, pages 1–17, 2021.

[33] Eman AlOmar, Mohamed Wiem Mkaouer, and Ali Ouni. Can refactoring be self-affirmed? an exploratory study on how developers document their refactoring activities in commit messages. In *2019 IEEE/ACM 3rd International Workshop on Refactoring (IWoR)*, pages 51–58. IEEE, 2019.

[34] Alex Bogart, Eman Abdullah AlOmar, Mohamed Wiem Mkaouer, and Ali Ouni. Increasing the trust in refactoring through visualization. In

2020 IEEE/ACM 4th International Workshop on Refactoring (IWoR), 2020.

[35] Eman Abdullah AlOmar, Anthony Peruma, Mohamed Wiem Mkaouer, Christian Newman, Ali Ouni, and Marouane Kessentini. How we refactor and how we document it? on the use of supervised machine learning algorithms to classify refactoring documentation. *Expert Systems with Applications*, 167:114176, 2021.

[36] Eman Abdullah AlOmar, Hussein AlRubaye, Mohamed Wiem Mkaouer, Ali Ouni, and Marouane Kessentini. Refactoring practices in the context of modern code review: An industrial case study at xerox. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 348–357. IEEE, 2021.

[37] Eman Abdullah AlOmar, Anthony Peruma, Christian D Newman, Mohamed Wiem Mkaouer, and Ali Ouni. On the relationship between developer experience and refactoring: An exploratory study and preliminary results. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, pages 342–349, 2020.

[38] Eman Abdullah AlOmar, Philip T Rodriguez, Jordan Bowman, Tianjia Wang, Benjamin Adepoju, Kevin Lopez, Christian Newman, Ali Ouni, and Mohamed Wiem Mkaouer. How do developers refactor code to improve code reusability? In *International Conference on Software and Software Reuse*, pages 261–276. Springer, 2020.

[39] Eman Abdullah AlOmar, Tianjia Wang, Raut Vaibhavi, Mohamed Wiem Mkaouer, Christian Newman, and Ali Ouni. Refactoring

for reuse: An empirical study. *Innovations in Systems and Software Engineering*, pages 1–31, 2021.

[40] Anthony Peruma, Khalid Almalki, Christian D. Newman, Mohamed Wiem Mkaouer, Ali Ouni, and Fabio Palomba. Tsdetect: An open source test smells detection tool. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, New York, NY, USA, 2020. Association for Computing Machinery.

[41] Anthony Peruma, Christian D. Newman, Mohamed Wiem Mkaouer, Ali Ouni, and Fabio Palomba. An exploratory study on the refactoring of unit test files in android applications. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, ICSEW'20, page 350–357, New York, NY, USA, 2020. Association for Computing Machinery.

[42] Anthony Peruma, Khalid Almalki, Christian D. Newman, Mohamed Wiem Mkaouer, Ali Ouni, and Fabio Palomba. On the distribution of test smells in open source android applications: An exploratory study. In *Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering*, CASCON '19, page 193–202, USA, 2019. IBM Corp.

[43] Sirine Gharbi, Mohamed Wiem Mkaouer, Ilyes Jenhani, and Montassar Ben Messaoud. On the classification of software change messages using multi-label active learning. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*, pages 1760–1767, 2019.

[44] Wiem Mkaouer, Marouane Kessentini, Adnan Shaout, Patrice Koligheu, Slim Bechikh, Kalyanmoy Deb, and Ali Ouni. Many-objective software remodularization using nsga-iii. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 24(3):1–45, 2015.

[45] Mohamed Wiem Mkaouer, Marouane Kessentini, Slim Bechikh, Kalyanmoy Deb, and Mel Ó Cinnéide. Recommendation system for software refactoring using innovization and interactive dynamic optimization. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 331–336, 2014.

[46] Mohamed Wiem Mkaouer, Marouane Kessentini, Slim Bechikh, Kalyanmoy Deb, and Mel Ó Cinnéide. High dimensional search-based software engineering: finding tradeoffs among 15 objectives for automating software refactoring using nsga-iii. In *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation*, pages 1263–1270, 2014.

[47] Mohamed Wiem Mkaouer, Marouane Kessentini, Slim Bechikh, Mel Ó Cinnéide, and Kalyanmoy Deb. On the use of many quality attributes for software refactoring: a many-objective search-based software engineering approach. *Empirical Software Engineering*, 21(6):2503–2545, 2016.

[48] Mohamed Wiem Mkaouer, Marouane Kessentini, Mel Ó Cinnéide, Shinpei Hayashi, and Kalyanmoy Deb. A robust multi-objective approach to balance severity and importance of refactoring opportunities. *Empirical Software Engineering*, 22(2):894–927, 2017.

[49] Rafi Almhana, Wiem Mkaouer, Marouane Kessentini, and Ali Ouni. Recommending relevant classes for bug reports using multi-objective search. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 286–295. IEEE, 2016.

[50] Anthony Peruma, Khalid Almalki, Christian D Newman, Mohamed Wiem Mkaouer, Ali Ouni, and Fabio Palomba. On the distribution of test smells in open source android applications: An exploratory study. In *Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering*, pages 193–202, 2019.

[51] Islem Saidani, Ali Ouni, and Mohamed Wiem Mkaouer. Improving the prediction of continuous integration build failures using deep learning. *Automated Software Engineering*, 29(1):1–61, 2022.

[52] Wajdi Aljedaani, Mona Aljedaani, Eman Abdullah AlOmar, Mohamed Wiem Mkaouer, Stephanie Ludi, and Yousef Bani Khalaf. I cannot see you—the perspectives of deaf students to online learning during covid-19 pandemic: Saudi arabia case study. *Education Sciences*, 11(11):712, 2021.

[53] Islem Saidani, Ali Ouni, and Wiem Mkaouer. Detecting skipped commits in continuous integration using multi-objective evolutionary search. *IEEE Transactions on Software Engineering*, 2021.

[54] Marwa Daaji, Ali Ouni, Mohamed Mohsen Gammoudi, Salah Bouktif, and Mohamed Wiem Mkaouer. Multi-criteria web services selection: Balancing the quality of design and quality of service. *ACM Transactions on Internet Technology (TOIT)*, 22(1):1–31, 2021.

[55] Nuri Almarimi, Ali Ouni, Moataz Chouchen, and Mohamed Wiem Mkaouer. csdetector: an open source tool for community smells detection. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1560–1564, 2021.

[56] Islem Saidani, Ali Ouni, Moataz Chouchen, and Mohamed Wiem Mkaouer. Bf-detector: an automated tool for ci build failure detection. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1530–1534, 2021.

[57] Oumayma Hamdi, Ali Ouni, Mel Ó Cinnéide, and Mohamed Wiem Mkaouer. A longitudinal study of the impact of refactoring in android applications. *Information and Software Technology*, 140:106699, 2021.

[58] Oumayma Hamdi, Ali Ouni, Eman Abdullah AlOmar, Mel Ó Cinnéide, and Mohamed Wiem Mkaouer. An empirical study on the impact of refactoring on quality metrics in android applications. In *2021 IEEE/ACM 8th International Conference on Mobile Software Engineering and Systems (MobileSoft)*, pages 28–39. IEEE, 2021.

[59] Islem Saidani, Ali Ouni, Mohamed Wiem Mkaouer, and Fabio Palomba. On the impact of continuous integration on refactoring practice: An exploratory study on travistorrent. *Information and Software Technology*, 138:106618, 2021.

[60] Lobna Ghadhab, Ilyes Jenhani, Mohamed Wiem Mkaouer, and Montassar Ben Messaoud. Augmenting commit classification by using fine-

grained source code changes and a pre-trained deep neural language model. *Information and Software Technology*, 135:106566, 2021.

[61] Fan Fang, John Wu, Yanyan Li, Xin Ye, Wajdi Aljedaani, and Mohamed Wiem Mkaouer. On the classification of bug reports to improve bug localization. *Soft Computing*, 25(11):7307–7323, 2021.

[62] Makram Soui, Mabrouka Chouchane, Narjes Bessghaier, Mohamed Wiem Mkaouer, and Marouane Kessentini. On the impact of aesthetic defects on the maintainability of mobile graphical user interfaces: An empirical study. *Information Systems Frontiers*, pages 1–18, 2021.

[63] Eman Abdullah AlOmar, Hussein AlRubaye, Mohamed Wiem Mkaouer, Ali Ouni, and Marouane Kessentini. Refactoring practices in the context of modern code review: An industrial case study at xerox. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 348–357. IEEE, 2021.

[64] Moataz Chouchen, Ali Ouni, Raula Gaikovina Kula, Dong Wang, Patanamon Thongtanunam, Mohamed Wiem Mkaouer, and Kenichi Matsumoto. Anti-patterns in modern code review: Symptoms and prevalence. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 531–535. IEEE, 2021.

[65] Xin Ye, Yongjie Zheng, Wajdi Aljedaani, and Mohamed Wiem Mkaouer. Recommending pull request reviewers based on code changes. *Soft Computing*, 25(7):5619–5632, 2021.

[66] Hussein Alrubaye, Deema Alshoaibi, Eman Alomar, Mohamed Wiem Mkaouer, and Ali Ouni. How does library migration impact software quality and comprehension? an empirical study. In *International Conference on Software and Software Reuse*, pages 245–260. Springer, 2020.

[67] Moataz Chouchen, Ali Ouni, Mohamed Wiem Mkaouer, Raula Gaikovina Kula, and Katsuro Inoue. Whoreview: A multi-objective search-based approach for code reviewers recommendation in modern code review. *Applied Soft Computing*, 100:106908, 2021.

[68] Moataz Chouchen, Ali Ouni, and Mohamed Wiem Mkaouer. Androlib: Third-party software library recommendation for android applications. In *International Conference on Software and Software Reuse*, pages 208–225. Springer, 2020.

[69] Nuri Almarimi, Ali Ouni, Moataz Chouchen, Islem Saidani, and Mohamed Wiem Mkaouer. On the detection of community smells using genetic programming-based ensemble classifier chain. In *Proceedings of the 15th International Conference on Global Software Engineering*, pages 43–54, 2020.

[70] Eman Abdullah AlOmar, Mohamed Wiem Mkaouer, and Ali Ouni. Toward the automatic classification of self-affirmed refactoring. *Journal of Systems and Software*, 171:110821, 2021.

[71] Islem Saidani, Ali Ouni, and Mohamed Wiem Mkaouer. Web service api anti-patterns detection as a multi-label learning problem. In *International Conference on Web Services*, pages 114–132. Springer, 2020.

[72] Bader Alkhazi, Andrew DiStasi, Wajdi Aljedaani, Hussein Alrubaye, Xin Ye, and Mohamed Wiem Mkaouer. Learning to rank developers for bug report assignment. *Applied Soft Computing*, 95:106667, 2020.

[73] Motaz Chouchen, Ali Ouni, Mohamed Wiem Mkaouer, Raula Gaikovina Kula, and Katsuro Inoue. Recommending peer reviewers in modern code review: a multi-objective search-based approach. In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion*, pages 307–308, 2020.

[74] Islem Saidani, Ali Ouni, Moataz Chouchen, and Mohamed Wiem Mkaouer. Predicting continuous integration build failures using evolutionary search. *Information and Software Technology*, 128:106392, 2020.

[75] Anthony Peruma, Khalid Almalki, Christian D Newman, Mohamed Wiem Mkaouer, Ali Ouni, and Fabio Palomba. tsdetect: an open source test smells detection tool. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1650–1654, 2020.

[76] Islem Saidani, Ali Ouni, Moataz Chouchen, and Mohamed Wiem Mkaouer. On the prediction of continuous integration build failures using search-based software engineering. In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion*, pages 313–314, 2020.

[77] Nuri Almarimi, Ali Ouni, and Mohamed Wiem Mkaouer. Learning to detect community smells in open source software projects. *Knowledge-Based Systems*, 204:106201, 2020.

[78] Islem Saidani, Ali Ouni, Mohamed Wiem Mkaouer, and Aymen Saied. Towards automated microservices extraction using muti-objective evolutionary search. In *International Conference on Service-Oriented Computing*, pages 58–63. Springer, Cham, 2019.

[79] Nuri Almarimi, Ali Ouni, Salah Bouktif, Mohamed Wiem Mkaouer, Raula Gaikovina Kula, and Mohamed Aymen Saied. Web service api recommendation for automated mashup creation using multi-objective evolutionary search. *Applied Soft Computing*, 85:105830, 2019.

[80] Makram Soui, Mabrouka Chouchane, Mohamed Wiem Mkaouer, Marouane Kessentini, and Khaled Ghedira. Assessing the quality of mobile graphical user interfaces using multi-objective optimization. *Soft Computing*, 24(10):7685–7714, 2020.

[81] Nasir Safdari, Hussein Alrubaye, Wajdi Aljedaani, Bladimir Baez Baez, Andrew DiStasi, and Mohamed Wiem Mkaouer. Learning to rank faulty source files for dependent bug reports. In *Big Data: Learning, Analytics, and Applications*, volume 10989, page 109890B. International Society for Optics and Photonics, 2019.

[82] Vahid Alizadeh, Marouane Kessentini, Mohamed Wiem Mkaouer, Mel Ocinneide, Ali Ouni, and Yuanfang Cai. An interactive and dynamic search-based approach to software refactoring recommendations. *IEEE Transactions on Software Engineering*, 46(9):932–961, 2018.

[83] Anthony Peruma, Mohamed Wiem Mkaouer, Michael J Decker, and Christian D Newman. An empirical investigation of how and why developers rename identifiers. In *Proceedings of the 2nd International Workshop on Refactoring*, pages 26–33. ACM, 2018.

[84] Makram Soui, Mabrouka Chouchane, Ines Gasmi, and Mohamed Wiem Mkaouer. Plain: Plugin for predicting the usability of mobile user interface. In *VISIGRAPP (1: GRAPP)*, pages 127–136, 2017.

[85] Ian Shoenberger, Mohamed Wiem Mkaouer, and Marouane Kessentini. On the use of smelly examples to detect code smells in javascript. In *European Conference on the Applications of Evolutionary Computation*, pages 20–34. Springer, Cham, 2017.

[86] Mohamed Wiem Mkaouer. Interactive code smells detection: An initial investigation. In *International Symposium on Search Based Software Engineering*, pages 281–287. Springer, Cham, 2016.

[87] Mohamed Wiem Mkaouer, Marouane Kessentini, Slim Bechikh, and Mel Ó Cinnéide. A robust multi-objective approach for software refactoring under uncertainty. In *International Symposium on Search Based Software Engineering*, pages 168–183. Springer, Cham, 2014.

[88] Mohamed Wiem Mkaouer and Marouane Kessentini. Model transformation using multiobjective optimization. In *Advances in Computers*, volume 92, pages 161–202. Elsevier, 2014.

[89] Mohamed W Mkaouer, Marouane Kessentini, Slim Bechikh, and Daniel R Tauritz. Preference-based multi-objective software modelling. In *2013 1st International Workshop on Combining Modelling and Search-Based Software Engineering (CMSBSE)*, pages 61–66. IEEE, 2013.

[90] T. Lu I. Chen, C. Yang and H. Jaygarl. Implicit social network model for predicting and tracking the location of faults. *IEEE International Computer Software and Applications Conference*, pages 136–143, 2008.

[91] N. A. Kraft S. K. Lukins and L. H. Etzkorn. Source code retrieval for bug localization using latent dirichlet allocation. *Working Conference on Reverse Engineering*, pages 155–164, 2008.

[92] H. Zhang J. Zhou and D. Lo. Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports. *International Conference on Software Engineering*, pages 14–24, 2012.

[93] Y. Wang et al. Bug localization via supervised topic modeling. *IEEE International Conference on Data Mining*, pages 607–616, 2018.

[94] Laura Moreno, John Joseph Treadway, Andrian Marcus, and Wuwei Shen. On the use of stack traces to improve text retrieval-based bug localization. *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 151–160, 2014.

[95] Xin Ye, Hui Shen, Xiao Ma, Razvan Bunescu, and Chang Liu. From word embeddings to document similarities for improved information retrieval in software engineering. In *Proceedings of the 38th international conference on software engineering*, pages 404–415, 2016.

[96] Shubham et al. Sangle. Drast - a deep learning and ast based approach for bug localization. *International Conference on Software Engineering*, 2020.

[97] Jinglei Zhang, Rui Xie, Wei Ye, Yuhan Zhang, and Shikun Zhang. Exploiting code knowledge graph for bug localization via bi-directional attention. In *Proceedings of the 28th International Conference on Program Comprehension*, pages 219–229, 2020.

[98] S. Khurshid R. K. Saha, M. Lease and D. E. Perry. Improving bug localization using structured information retrieval. *International Conference on Automated Software Engineering*, pages 345–355, 2013.

[99] Anh Tuan Nguyen, Tung Thanh Nguyen, Jafar Al-Kofahi, Hung Viet Nguyen, and Tien N Nguyen. A topic-based approach for narrowing the search space of buggy files from a bug report. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, pages 263–272. IEEE, 2011.

[100] Chu-Pan Wong, Yingfei Xiong, Hongyu Zhang, Dan Hao, Lu Zhang, and Hong Mei. Boosting bug-report-oriented fault localization with segmentation and stack-trace analysis. *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 181–190, 2014.

[101] Bunyamin Sisman and Avinash C. Kak. Incorporating version histories in information retrieval based bug localization. *2012 9th IEEE Working Conference on Mining Software Repositories (MSR)*, pages 50–59, 2012.

[102] Shanto Rahman, Kishan Kumar Ganguly, and Kazi Sakib. An improved bug localization using structured information retrieval and version history. *2015 18th International Conference on Computer and Information Technology (ICCIT)*, pages 190–195, 2015.

[103] Rashina Hoda. Socio-technical grounded theory for software engineering. *IEEE Transactions on Software Engineering*, pages 1–1, 2021.

[104] Adrian Schroter, Adrian Schröter, Nicolas Bettenburg, and Rahul Premraj. Do stack traces help developers fix bugs? In *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*, pages 118–121, 2010.

[105] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. Distributed representations of words and phrases and their compositionality. In *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 2*, page 3111–3119, Red Hook, NY, USA, 2013. Curran Associates Inc.

[106] Jeffrey Pennington, Richard Socher, and Christopher Manning. GloVe: Global vectors for word representation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543, Doha, Qatar, October 2014. Association for Computational Linguistics.

[107] Michael L. Collard, Michael John Decker, and Jonathan I. Maletic. srcml: An infrastructure for the exploration, analysis, and manipulation of source code: A tool demonstration. In *2013 IEEE International Conference on Software Maintenance*, pages 516–519, 2013.

[108] Michael Hucka. Spiral: splitters for identifiers in source code files. *Journal of Open Source Software*, 3(24):653, 2018.