5-2022

Taxonomy of Software Readability Changes

Stephen J. Cook
sjc5897@rit.edu

Taxonomy of Software Readability Changes

by

Stephen J. Cook

A thesis submitted in partial fulfillment of the
requirements for the degree of
Master of Science
in Software Engineering

B. Thomas Golisano College of Computing and
Information Sciences
Rochester Institute of Technology

[May 2022]

MS IN SOFTWARE ENGINEERING
ROCHESTER INSTITUTE OF TECHNOLOGY
ROCHESTER, NEW YORK

CERTIFICATE OF APPROVAL

MS DEGREE THESIS

The MS degree thesis of Stephen J. Cook
has been examined and approved by the
thesis committee as satisfactory for the
thesis required for the
MS degree in Software Engineering

____
Christian D. Newman, Thesis Advisor

____
J. Scott Hawker, External Chair

____
Mohamed Wiem Mkaouer

____
Date

Abstract

Software readability has emerged as an important software quality metric. Numerous pieces of research have highlighted the importance of readability. Developers generally spend a large amount of their time reading and understanding existing code, rather than writing new code [1] [2]. By creating more readable code, engineers can limit the mental load required to understand specific code segments [3]. With this importance established, research has been done into how to improve software readability. This research looked for ways of measuring readability, how to create more readable software, and how to potentially improve readability. While some research has examined the changes developers make, their use of automatic source code analysis may miss some aspects of these changes. As such, this study conducted a manual review of software readability commits to identify what changes developers tend to make.

In this study, we identified 1,782 potential readability commits for 800 open-source Java projects, by mining keyword patterns in commit messages. These commits were then reviewed by human reviewers to identify the changes made by the developers. The observations made by the reviewers were then reviewed for trends, from which several categories would be established. These categories would be further reviewed for additional trends, developing a taxonomy of readability changes. Overall, this research looked at 314 changes from 194 commits across 154 unique projects.

This study shows the developers' actions when improving software readability, identifying the common trends of method extraction, identifier renaming, and code formatting, supported by existing research. In addition, this research presents less observed trends, such as code removal or keyword modification, which were changes not seen in other research. Overall, this work provides a taxonomy of the trends seen, identifying high level trends as well as subgroups within those trends.

Acknowledgments

As with any major work, several people provide their support and knowledge. I would like to extend my thanks to all those who supported me through the creation of this work.

I would first like to thank my advisor, Dr. Christian Newman, for their support and insight into this project. Without your contributions to this work, it would likely not have been finished. I would like to extend my deepest thanks to you, it was a joy to work with you. I would also like to extend my thanks to Anthony Peruma, who work paved the way for mine. Not only was the initial database created and provided by Anthony, but I also gained a lot of insight into the process of software research both directly and indirectly from Anthony. Finally, I would like to extend my thanks to the members of my committee for their time and additional insight.

I would also like to thank those who supported me in less direct ways, that being my friends and family. While the academic support provided was important, the moral support provided by them was equally as important. To each one of you, I say thank you.

To my mother, Doreen, who supported me throughout this endeavor

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Software readability has emerged as a key aspect of software quality. Software readability influences the effort a developer must make to understand a piece of existing code. Research has shown that poor source code readability can significantly increase a developers' cognitive load [4]. With most of a developer's time spent trying to comprehend source code, improving software readability can improve developer efficiency [2] be it the creation of new code or the maintenance of existing code. The challenge comes from improving the readability, as readability is a subjective quality.

A focus has been placed on the research of software readability as a quality metric. Books have been written on creating high-quality, highly readable software [2]. Numerous other researchers have attempted to create models which can quantify software readability and detect improvements [5] [6] [7] [8] [9]; however, some research has cast doubt on these models' effectiveness [3] [10]. With this vast amount of research, most of the focus has been placed on the creation of new software. While research shows that this focus at the start of the cycle does create more readable software [11] it does not directly guide those who wish to improve existing code bases. In order to better understand how to make software more readable, attention needs to be brought to how developers can improve readability. By looking more directly at the changes made by developers with the intention of improving readability, recommendations and strategies can be identified. While this has been done to some extent in other research, such as Fakhoury et al. [3], they analyzed the commits using source code analysis tools. By taking a more direct look at these commits using a manual review based in Grounded Theory [12], other trends maybe identified which are missed by these automatic tools.

The goal of this paper is to identify the types of changes developers make when attempting to improve software readability. This is accomplished first by creating a dataset of commits which claim they improve readability. These commits are then reviewed by manual reviewers, who identify the changes made within readability commits. These observations by the reviewers are then examined for common trends across the projects. These trends are then further analyzed for potential trends within them. Finally, these observations and trends are used to produce the major contribution of this paper, the taxonomy of readability changes.

The rest of this paper is structured as follows: Chapter 2 covers the overall research objectives, including the motivations, contributions, and questions. Chapter 3 outlines the methodology applied to achieve the research objectives, highlighting the data collection and analysis process. Chapter 4 covers the proposed taxonomy, going into detail on the categorizations and their natures. Chapter 5 addresses the research results, exploring in detail the data seen in the research. Chapter 5 also provides some discussion of this data. Chapter 6 explores some of the other work within the field of readability research. Chapter 7 identifies some issues with the research which may affect its validity. Chapter 8 discusses the potential future work and improvements. Finally, Chapter 9 concludes this paper and highlights the takeaways.

Chapter 2

Research Objectives

2.1 Research Motivation

Software maintenance tasks have become more time-consuming with the growing scope of software projects. The increased scope means that the code base often is larger and more complex. This creates more demand on the developer performing the maintenance task, taking more time to understand the code. An important way of mitigating this issue is having code that is easier for the developer to read and understand; however, the challenge of readability research is its subjective nature. Determining how to improve software readability is difficult. While research has been done into the creation of readability metrics [5], [6], [7] [8] [9] and into the types of changes developers make [3] [11], their partially automated approaches in research may miss some nuance in the changes made by developers. **This research aim to identify changes developers make when attempting to improve software readability.** By applying Grounded Theory to a manual review of readability changes, this study presents a taxonomy for these readability changes. In addition, this research acts to both confirm and augment prior research on the types of changes that constitute readability changes.

2.2 Research Contributions

Our primary contribution through this study is the created taxonomy for software readability improvements. This was achieved by:

- Gathering a data set of identifiable readability changes from a set of refactoring commits on engineered open-source projects.
- Providing a set of manual observations on these change and observations about trends within those changes.
- Presenting a developed readability taxonomy based on the observations.

As this research represents a first step in the creation of a readability change taxonomy, the existing data and set of observations is publicly available on GitHub, along with the scripts used in the mining of data [1].

2.3 Research Questions

To pursue our research goals, two research questions were created:

- **RQ1: Which keyword patterns were most highly correlated with readability commits?** By creating keyword patterns, we hope

[1]https://github.com/sjc5897/readability_taxonomy

to leverage them in the detection of readability commits. In addition, we tracked the number of commits that contain these patterns and precision in detecting readability commits. The goal is to report this precision to assist in future detection research. The analysis of keywords helps us identify self-admitted readability improvements, which is key in understanding what changes developers make. By more closely analyzing their effectiveness, we can improve our detection of self-admitted readability improvements, making our readability analysis more effective.

- **RQ2: What code changes do developers make when they admit that they are trying to improve code readability?** By exploring the changes made, we present a taxonomy of these changes. Highlighting common trends seen within the commits while also exploring the subtypes of the most common trends. The goal is to report the observed trends within the detected commits from a high level. This question directly targets the goal of the paper, looking directly at what changes are made and how commonly these changes are made.

Chapter 3

Methodology

The research methodology of this project consisted of two stages: data collection and data analysis. In the data collection stage, a set of potential readability commits were gathered. This stage was done using an automatic process to identify and log potential readability commits from a set of refactoring commits, using keyword mining. The potential readability commits were reviewed in the data analysis stage. This review was performed by a human reviewer. The reviewer was first asked to confirm if the change was a readability change. When a change was identified, it was logged, and observations of the changes were made. The observations made by the reviewer were then reviewed to identify trends in the data. Figure 3.1 highlights the methodology.

3.1 Data Collection

The initial stage of this project was the data collection stage. The goal of this phase is the gathering of potential open-source readability commits which

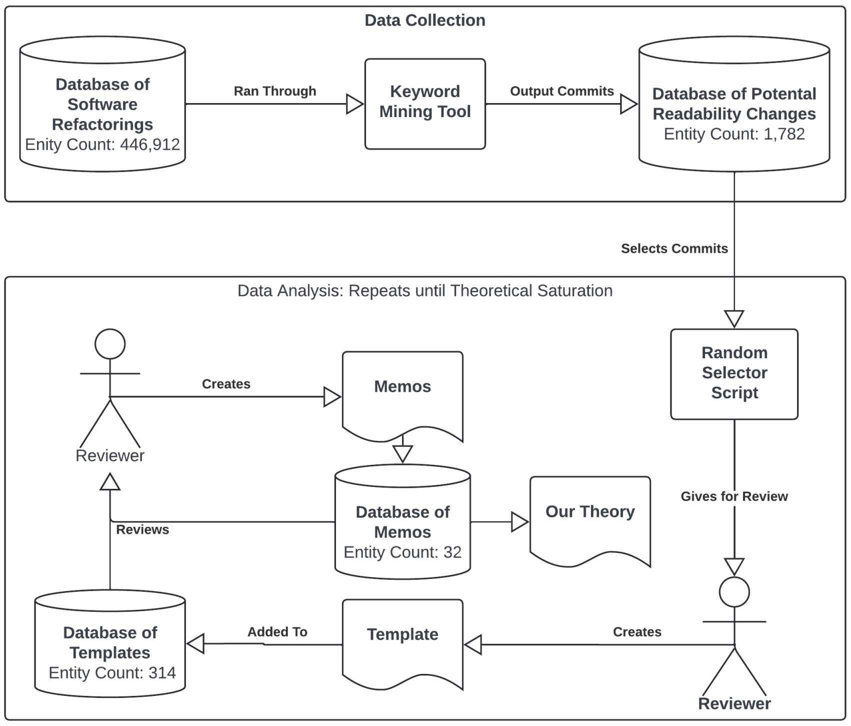


Figure 3.1: An overview of the developed methodology

could be reviewed. To do this, a simple Python script using Pandas was developed to help mine commits for keyword patterns. Three keyword patterns were targeted: *'readab.*'*, *'understand.*'* and *'clean.* up'*. The targeting of *'readab.*'* and *'understand.*'* directly targeted words associated with readability terms. In addition, E.A. AlOmar et al. were consulted for other patterns. In this paper, E.A. AlOmar et al. performed research into how refactors were documented [13]. As part of this research, they identified several patterns used across commit messages. While a number of these patterns could have been used for this research, such as *'simplif.** or *"improv.*'*, the *'clean.* up'* keyword pattern was chosen as it was the broadest [13]. These patterns were used to identify commit messages which indicate readability improvements.

Once the tool was created to identify keyword patterns from commit messages, it was applied to an existing database of Java software refactors. The targeted database for this research was provided by the authors of Peruma et al [14]. This database is a random sample of 800 projects from Munaiah et al's engineered projects database [15]. The commits from 800 projects were then run through *RefactoringMiner* by Peruma et al. to identify specifically refactoring commits [14]. It is this segment of data that was used for our research. This gave us a set of commits in which refactoring occurred from a set of engineered open-source software projects. Refactoring commits were chosen as the focus because refactors are typically done to improve internal code qualities such as readability, rather than creating new behaviors or features. Since we are concerned with internal code quality improvement via readability, this focus helped eliminate some potential noise from data collection. From this dataset, 1,782 potential readability commits were identified. These commits would make up the set of commits we sampled from.

3.2 Data Analysis

With the set of readability commits extracted from the database, a method for data analysis was created. This process required manual review by a human reviewer. This review would leverage Grounded Theory, specifically the cycle of data collection, coding, and constant comparison, to develop a theory. From a high level, a sample change from the database would be selected. A reviewer

Table 3.1: Review Template

Is this a Readability Change:	[Y/N]		
	Description of Code:		
		Describe the Code Change:	[Free Response]
		Describe the Code Functionality:	[Free Response]
		Keywords:	[Free Response]
	Description of Commit:		
		Describe the Commit Message in Terms of Readability:	[Free Response]
	Location Information		
		Files:	[Free Response]
		Old Line Numbers:	[Free Response]
		New Line Numbers:	[Free Response]

would perform a review, writing down observations of the change and tagging it using keywords. These observations would be reviewed in regular intervals, and categories would be formed. These categories represented the theoretical readability taxonomy. This process would repeat until we hit theoretical saturation at 314 changes observed. Our taxonomy in Chapter 4 represents the final theory generated from these observations.

3.2.1 Collecting Commits

In the Grounded Theory cycle, data needs to be extracted. Since we had collected a set of potential readability changes, our data collection would consist of random sampling from this database. To help with this, the initial data mining script was expanded. The script would segment the sample database by project, randomize the list of projects, and iterate through that list, selecting one commit to present for review. This was to prevent an over-representation of one project in the data set. These changes would then be presented to the reviewer for review. With this sampling, the final set of observations would be made from 194 commits from 154 different projects, with at most 4 commits seen from an individual project.

With each sample commit, it was presented to the reviewer. The reviewer would first determine if the commit was in fact a readability commit. Since it was dubious if the keywords were accurate in the goal of discovering readability changes, the reviewer was asked if the commit was indeed a readability change. The reviewer was given a few criteria to determine if a readability change occurred. First, priority was given to the commit message. If the commit message directly stated a change was done for readability improvements, regardless of if the reviewer agreed, it was considered a readability improvement. If this change was not found in the code, it was marked as a non-readability commit. If the commit message was vague, not calling out specific changes, the reviewer would manually review all the changes to identify a readability change. At this point, a readability change became more subjective based on reviewer opinion. The general guidance was a readability change would be a change to the source code which did not significantly impact observable code behavior. Finally, changes to test files were disregarded in manual review. This was done to focus on improving production code. Changes to only test files would be marked as non-readability commits.

3.2.2 Grounded Theory

Once a readability commit was identified, the reviewer was asked a series of curated questions about the change. Table 3.1, shows the question tree given to a reviewer. The first set were questions about the code itself. The reviewer was asked to describe the change and the code's functionality. In addition, the reviewer was asked to provide a set of keywords to assist in identification and search later. The reviewer was then asked to describe the commit message, being asked to describe how the developer related the change to readability within the commit. Finally, the reviewer was asked to provide location details, such as files and line numbers. These observations were stored in our database as entities called templates. The templates were automatically associated with commits by our Python script.

Each template would represent one change type per commit. This means that if 5 function extractions were done within one commit, this would be logged as one change type, thus given one template. A commit could have different types of changes, and these were generally kept separate. If a commit had both function extractions and renames occur, these would be logged as two different changes and given two different templates, one for each. This was done to prevent an over-saturation from one large refactoring commit or change type; however, change types were kept separate to distinguish them from each other. This set up means while we looked at 314 different changes, these changes only came from 194 commits. This gives us an average of 1.62 readability changes per commit. This process would provide us with a set of observations which we could analyze for trends. The templates and tags represented our attempt at coding the data [12].

Table 3.2: Example of Template: ID-24

Is this a Readability Change:	Y		
	Description of Code:		
		Describe the Code Change:	Extracted numerous code blocks into its own class, SecuritiesTables. These extracted code blocks were created as descriptive name functions within the class
		Describe the Code Functionality:	Manages a code concept called SecuritesTables
		Keywords:	function extraction,class extraction
	Description of Commit:		
		Describe the Commit Message in Terms of Readability:	Directly mentions that the extraction of the class improves readability
	Location Information		
		Files:	name.abuchen.portfolio.ui/src/name/abuchen/portfolio/ui/views/SecuritiesTable.java; name.abuchen.portfolio.ui/src/name/abuchen/portfolio/ui/views/SecurityListView.java
		Old Line Numbers:	Whole file
		New Line Numbers:	New file created

Table 3.3: Example of Memo: ID-1

ID	Memo Message	Keywords	Templates-Associated	Memos-Associated
1	Blocks of code are being extracted into separate functions, usually with the function given a descriptive identifier name	function extraction	24	-

In line with Grounded Theory's constant comparison and memoing [12], in regular periods throughout the templating stage, usually once a week, the existing observations would be analyzed by a reviewer. Similarly, to how observations of changes were made with templates, observations of trends were also made. These observations are called memos. A memo consists of a message and an association to templates and would also be stored in the database. As more and more memos were created, eventually observations would be made about the existing memos, creating even more memos. Table 3.2 and Table 3.3 show an example template and example memo respectively. These memos would form the backbone of theory development, as the identified categories from the taxonomy would be derived from the memos. Ideally, the memos should have been created more often than once a week; however, lack of available resources made this somewhat of a necessity.

This process of sampling, coding and comparison was repeated until theoretical saturation was achieved. Theoretical saturation is the point when more observations do not modify the developed theory [12]. The difficulty with readability research is that it may have taken a very long time to achieve theoretical saturation, as a very diverse number of changes could be made. As such, we targeted 317 changes as the sample size, which is the sample size that achieves 95% confidence with 5% margin of error on the population of 1,782. This gave us a target number of observations to make.

Chapter 4

Taxonomy

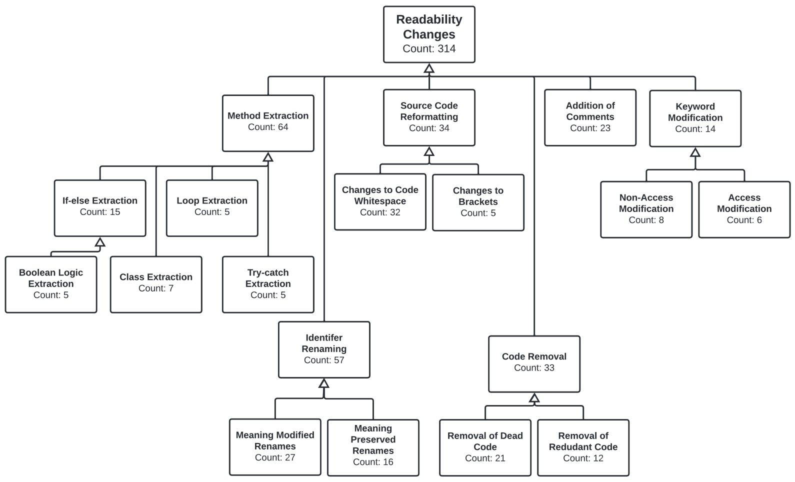

Readability Changes Count: 314
Method Extraction Count: 64
Source Code Reformatting Count: 34
Addition of Comments Count: 23
Keyword Modification Count: 14
If-else Extraction Count: 15
Loop Extraction Count: 5
Changes to Code Whitespace Count: 32
Changes to Brackets Count: 5
Non-Access Modification Count: 8
Access Modification Count: 6
Boolean Logic Extraction Count: 5
Class Extraction Count: 7
Try-catch Extraction Count: 5
Identifer Renaming Count: 57
Code Removal Count: 33
Meaning Modified Renames Count: 27
Meaning Preserved Renames Count: 16
Removal of Dead Code Count: 21
Removal of Redudant Code Count: 12

Figure 4.1: Proposed Readability Taxonomy

This section provides an overview of the produced readability taxonomy. This taxonomy categorizes the trends seen within identified readability changes. The overall taxonomy is represented in Figure 4.1. At the top is the Readability Changes group, representing the totality of the examined changes. The next level represents the six main categories which were identified. These categories are the most seen trends in the review. The layer under them is their identified subgroups, trends seen within the larger trends. The rest of this chapter is divided into six sections, each explaining the six main trends identified along with their subgroups. In addition, examples of each action are provided for further understanding.

4.1 Method Extraction

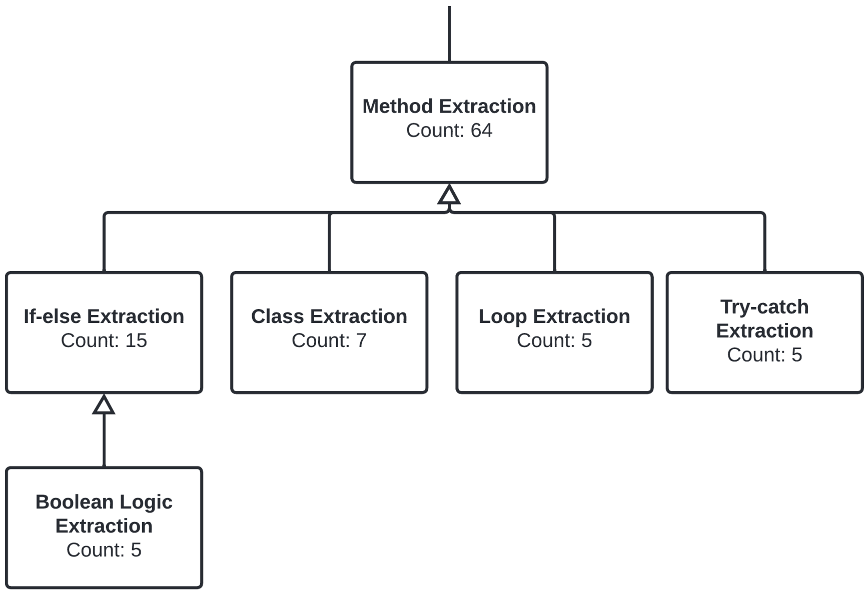
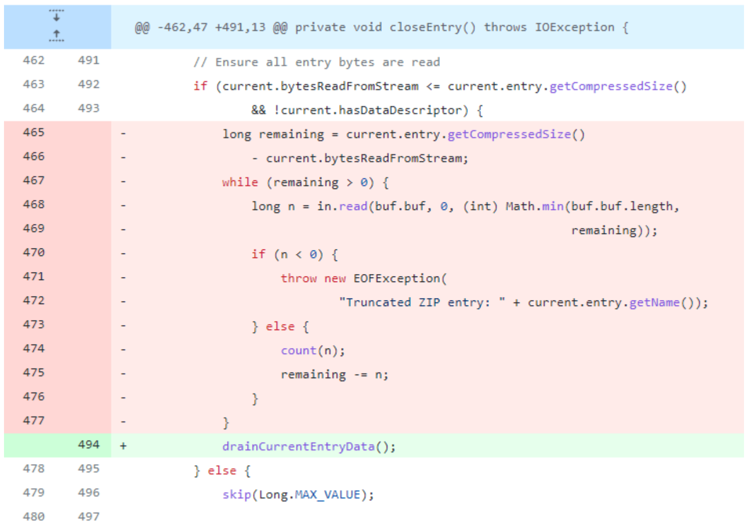


Figure 4.2: Method Extraction Taxonomy

The largest readability change type seen was method extraction. This categorization is identical to the common software refactoring Extract Function [16]. In method extraction, a code segment is removed from an existing block of code and added to a newly created method. The code segment is then replaced with a call to the newly created method. This new method is given a unique identifier, which can provide additional information about the method to the user. Typically, the extracted code is not modified, simply moved; however, it is still considered a method extraction if the original code segment is modified.

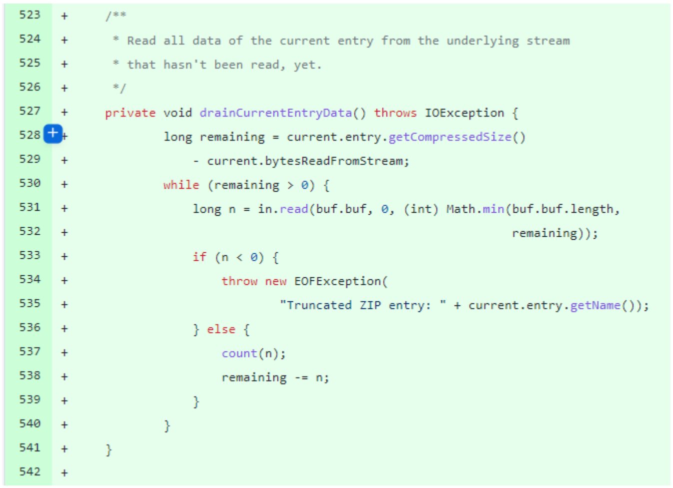


```
@@ -462,47 +491,13 @@ private void closeEntry() throws IOException {
          // Ensure all entry bytes are read
          if (current.bytesReadFromStream <= current.entry.getCompressedSize()
                  && !current.hasDataDescriptor) {
-             long remaining = current.entry.getCompressedSize()
-                 - current.bytesReadFromStream;
-             while (remaining > 0) {
-                 long n = in.read(buf.buf, 0, (int) Math.min(buf.buf.length,
-                                                             remaining));
-                 if (n < 0) {
-                     throw new EOFException(
-                             "Truncated ZIP entry: " + current.entry.getName());
-                 } else {
-                     count(n);
-                     remaining -= n;
-                 }
-             }
+             drainCurrentEntryData();
          } else {
              skip(Long.MAX_VALUE);

```

Figure 4.3: Example of Method Extraction Operation

Apache's Common-Compress Project highlights an example of the method extraction operation. Figure 4.3 shows a unified diff of the extraction. Here, a while loop and some setup code are removed and replaced with the function *drainCurrentEntryData*. Figure 4.4 shows the *drainCurrentEntryData* method declaration. This is a newly created method, with the body being the exact same as the removed code from Figure 4.3. In addition, the developer added a meaningful identifier and comments to the method declaration, which further supports a reader's understanding of the new method and code segment. The commit message for this change directly states that this change occurred to improve readability, and method extractions were performed in other places within the commit.



```
+    /**
+     * Read all data of the current entry from the underlying stream
+     * that hasn't been read, yet.
+     */
+    private void drainCurrentEntryData() throws IOException {
+        long remaining = current.entry.getCompressedSize()
+            - current.bytesReadFromStream;
+        while (remaining > 0) {
+            long n = in.read(buf.buf, 0, (int) Math.min(buf.buf.length,
+                                                         remaining));
+            if (n < 0) {
+                throw new EOFException(
+                        "Truncated ZIP entry: " + current.entry.getName());
+            } else {
+                count(n);
+                remaining -= n;
+            }
+        }
+    }
+
```

Figure 4.4: The *drainCurrentEntryData* method declaration

The method extraction change type can be further divided into four subcategories. These four categories are based on the trends seen in the type of code being extracted, mostly different control blocks. These groups are: if-else extraction, class extraction, loop extraction, and try-catch extraction. These categories were identified in a top-down manner. This means that method extraction was first identified, and these trends emerged from there. As a result, not all method extractions fit into one of the four categories, as with classifications seen later in this chapter.

The most common method extraction subcategory is the extraction of if-else ladders. In this category, part, or all, of an if-else ladder is extracted out to an external function. If-else extraction specifically includes the extraction of the if statements. While Figure 4.3 shows the body of an if statement being extracted, it does not fit into this category because the if statement is not moved. This category is unique among the method extractions, as it also has a subcategory, Boolean extraction. In Boolean extraction, the Boolean logic of an if statement, meaning the expression, is extracted to a function which returns the evaluation. This is distinct from if-else extraction, as it does not move the if-else ladder, moving just the expression.

The next most common type of method extraction is class extraction. This change is like two types of common software refactors. It can either be like a true class extraction, where a new class is created out of a set of behaviors, or like a move method, where a method is moved to an existing class [16]. In addition, this categorization does not focus on other class members, such as attributes. While there is likely an extraction of other class members, the observations only noted the movement of functions. Based on this observation, class extraction is set as a type of method extraction; however, this change could easily become its own category with further research.

The next two subcategories of method extraction are simpler than the previous two. The first of these is loop extraction, which is very similar to the if-else extraction. In this category, an entire looping block is extracted to a newly created function. The example provided in Figure 4.3 and 4.4 is in addition an example of loop extraction, as it extracts an entire while loop. Similarly, try-catch extraction extracts all of a try-catch block to an external function

4.2 Identifier Renames

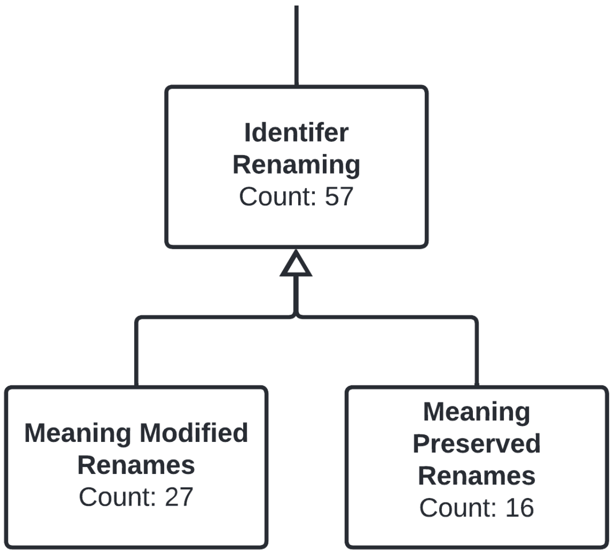
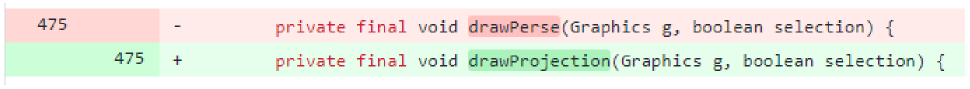


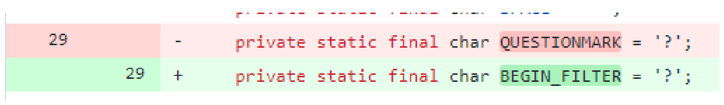
Figure 4.5: Identifier Rename Taxonomy



```
475     -     private final void drawParse(Graphics g, boolean selection) {
    475 +     private final void drawProjection(Graphics g, boolean selection) {
```

Figure 4.6: Example of Identifier Rename Operation

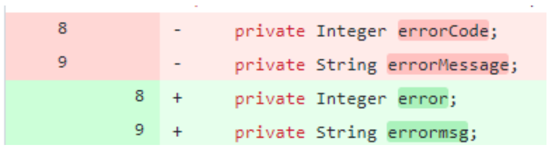
The second-largest category of readability changes is identifier renames. Figure 4.5 shows the taxonomy of renames. This change occurs whenever any software identifier, such as a function identifier or class identifier, is changed in any way. Software identifier names give the developer a great opportunity to provide valuable information to a reader without being constrained by syntax. An example of an identifier rename is shown in Figure 4.6, the Casmi developer renamed the function identifier of *drawParse* to *drawProjection*. Identifier renames fall within two categories, focusing on the change of the identifiers meaning. The meaning of the identifier can either be preserved (not changed) or modified (changed).



```
29     - private static final char QUESTIONMARK = '?';
    29 + private static final char BEGIN_FILTER = '?';
```

Figure 4.7: Example of Identifier Rename: Meaning Modified

The meaning of an identifier is modified if the meaning is: broadened, narrowed, or completely changed. When the meaning of an identifier is broadened, one or more of the terms in the identifier is replaced with a hypernym. A hypernym is simply a term that is higher on a semantic tree. If the word blue was changed to color, it would broaden the meaning, as color is a hypernym of the word blue. When the meaning of an identifier is narrowed, one or more of the terms is replaced with a hyponym. A hyponym is the opposite of a hypernym; it is a word lower on a semantic tree. Finally, the meaning can be completely changed by replacing words to new, unrelated words. While these three categories could become subcategories in themselves, it was hard to determine if broadening, narrowing or complete change occurred from the observations, more that the meanings were changed and generally these were the segmentation. To categorize them, a much more thorough analysis of each rename would need to be done, which would distract from other observations. The example provided in Figure 4.7 shows an example of a modified meaning rename. In this example, the identifier is changed from *'QUESTIONMARK'* to *'BEGIN_FILTER'*. This completely changes the meaning of the identifier to the reader, going from the actual value the identifier represents, a '?' symbol, to the use of the simple in context, the start of a filter string.



```
- private Integer errorCode;
- private String errorMessage;
+ private Integer error;
+ private String errormsg;
```

Figure 4.8: Example of Identifier Rename: Meaning Preserved

The simpler of the two to identify is if the meaning of the identifier is preserved. This category stands opposed to the other category, as the meaning cannot be preserved and modified. Generally, an identifier is considered to have its meaning preserved if a rename operation occurred, but the meaning was not modified. A few situations occurred that fit this definition. First were situations where the identifier separators were changed, such as a move from camel case to underscores. Second was a collapse or expansion of abbreviations. The example in Figure 4.8 highlights this type of change. In the rename for *errorMessage*, the word message was collapsed into a known abbreviation, *msg*. The third situation was fixing typos or misspellings, as these are just corrective actions. Finally, and most complex, is the use of synonyms which still preserve the overall meaning of the identifier. Similarly, to the previous category, these different types could make up another layer of the taxonomy, but with the way renames were coded it is hard to tell how prevalent these observations are in the data.

4.3 Source Code Reformatting

The third largest readability change category is source code reformatting. The taxonomy is highlighted in Figure 4.9. In source code reformatting, the developer changed the formatting of the code base without directly editing any of

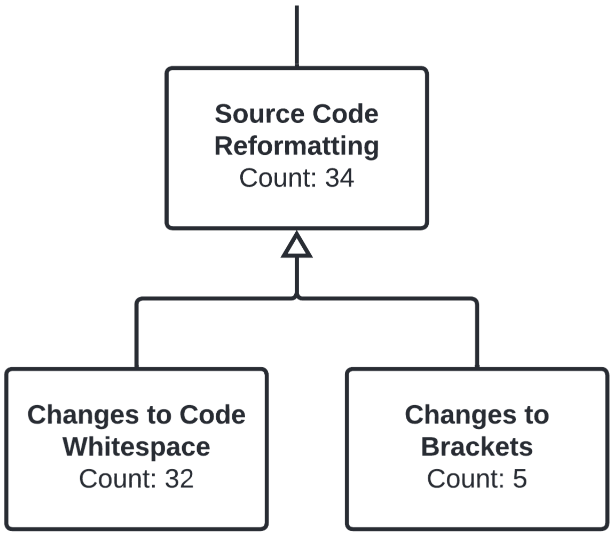
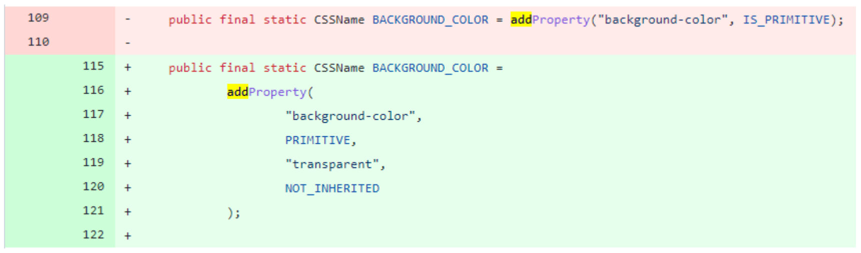


Figure 4.9: Taxonomy of Source Code Reformatting

the code. A simple example can be found within the Flying Saucer Project. In Figure 4.10 the before and after of the code change is shown. Here, the developer reformatted the function call, mostly using newlines to space out the function's parameters. The developer directly stated that this change would improve the readability of the software. While the addition of parameters modifies the code slightly, this change still highlights a common type of code reformatting preformed.



```
-     public final static CSSName BACKGROUND_COLOR = addProperty("background-color", IS_PRIMITIVE);
-
+     public final static CSSName BACKGROUND_COLOR =
+             addProperty(
+                     "background-color",
+                     PRIMITIVE,
+                     "transparent",
+                     NOT_INHERITED
+             );
+
```

Figure 4.10: Example of Source Code Reformatting

This category can then be subdivided into the two main methods of code reformatting. The largest of the two is changes to the code white space. With this change, the developer uses different types of white space characters to modify the spacing of the code. These white space characters include spaces, new lines, and tabs. Figure 4.10 is also an example of this subcategory, as it uses new lines to spread out the function call. White space source code formatting would often occur as part of cleanup operations.

The other category of source code formatting is editing of brackets in code. Any formatting change to the brackets in the code, such as adding curly braces or parentheses, is of this group. The most common example of this change is the addition of curly brackets to one line if-statements that do not have them. This change most often occurs with edits to white space; however, a couple of examples were seen without white space modification.

4.4 Code Removal

The fourth most common category of change for readability was the removal of code. With code removal, the developer deletes code that is no longer needed. While code removal can be done without the intent of improving readability, there are situations where the removal of code is done to improve the readability. The example, in Figure 4.12, shows how code removal can be used to improve readability. Before, the developer stored the output of *getClassPath* in the *cp* variable. Since the *cp* variable is only used once in line 199, the output storage is unneeded and can be removed. As seen at the bottom of the image, the adding of *cp* was replaced with a call to *getClassPath*.

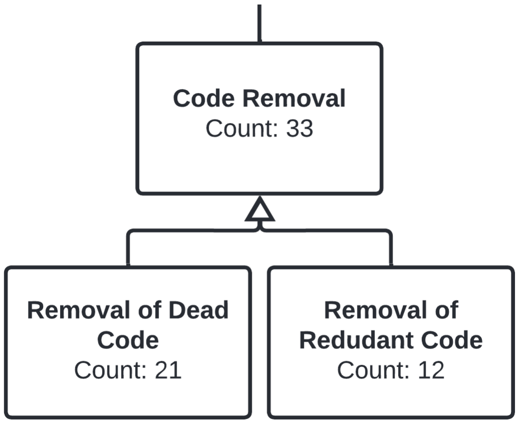
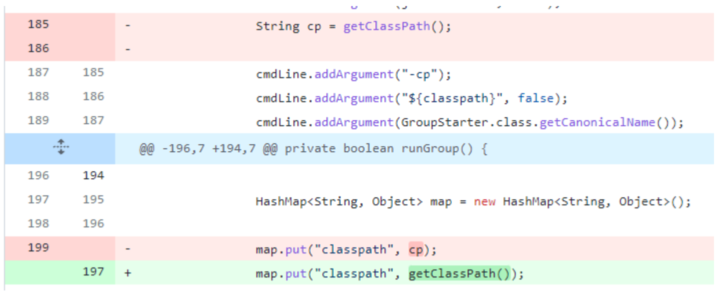


Figure 4.11: Taxonomy of Code Removal



```
-               String cp = getClassPath();
-
                cmdLine.addArgument("-cp");
                cmdLine.addArgument("${classpath}", false);
                cmdLine.addArgument(GroupStarter.class.getCanonicalName());
    @@ -196,7 +194,7 @@ private boolean runGroup() {

                HashMap<String, Object> map = new HashMap<String, Object>();

-               map.put("classpath", cp);
+               map.put("classpath", getClassPath());
```

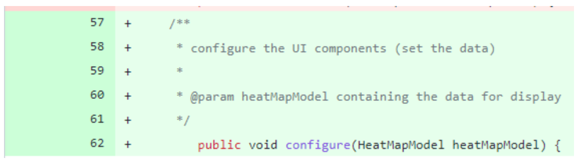
Figure 4.12: Example of Code Removal

Overall, the removal of code has been added to this taxonomy as it can be done to improve software readability.

This category can be broken into two types of removal. The first is the removal of dead code. In this definition, dead code covers three types of code segments. The first is code that is never executed. This can be an unused method or a code segment under a control statement that cannot be triggered. The second is executed code, but it does not modify the system and its output is not used. An example would be a function which counts the characters in a string, but the code never uses that count. The final type is code segments which are completely commented out.

The other category is the removal of redundant code. This category is distinct from the other in that the code does have some effect on the state of the system and is used; however, the developer has deemed it redundant in some way. There are a number of examples of this type of removal, but Figure 4.12 directly shows an example. Since the storage variable *cp* is used, but it is somewhat redundant to store the method output, it is the removal of redundant code. Other examples of this type of removal include the removal of function parameters or unnecessary attribute declarations.

4.5 Addition of Comments



```
+     /**
+      * configure the UI components (set the data)
+      *
+      * @param heatMapModel containing the data for display
+      */
+         public void configure(HeatMapModel heatMapModel) {
```

Figure 4.13: Example of Addition of Comments

The fifth category of readability change is the addition of comments by the developer. This category focuses on any in code documentation being added or modified, be it inline comments, multi-line comments or Javadoc. Adding comments to the source code can aid the reader in their understanding. In Figure 4.13, the simple addition of Javadoc as comments improves the software's readability. More research into the taxonomy of software comments could improve this sections taxonomy.

4.6 Keyword Modification

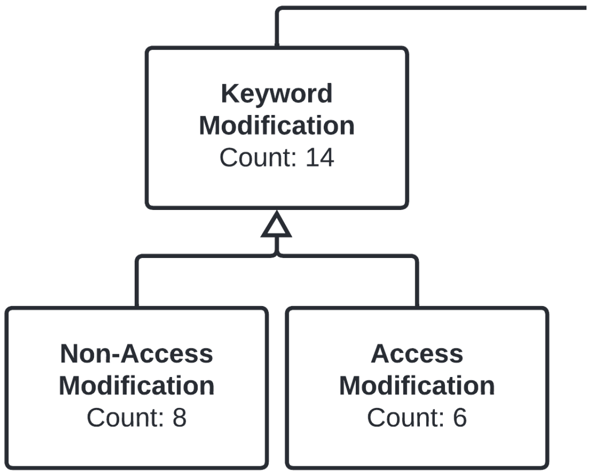
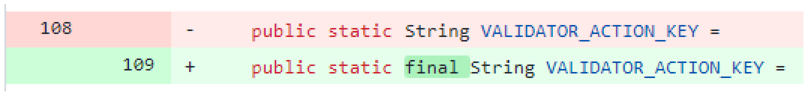


Figure 4.14: Taxonomy of Keyword Modification



```
-     public static String VALIDATOR_ACTION_KEY =
+     public static final String VALIDATOR_ACTION_KEY =
```

Figure 4.15: Example of Keyword Modification

The final category identified within this taxonomy is the keyword modification category. In this category, the developer changes the keywords in the software to help convey the purpose of a software entity. This includes the changing, addition, or removal of certain keywords. In Figure 4.15 an example of this type of change is shown. The developer of this commit added the final keyword to the static attribute declaration. While this change does influence the behavior of the code, preventing the variable's modification. The final keyword in this example provides the context for the developer that this variable does not change.

This category is separated into two categories, based on how Java divides up keywords. The first category is the change to access modifiers. This means the changing, adding or removing of keywords such as public, private, or protected. The most common change of this type is the addition of access modifiers. In a few cases, classes or methods did not have a specified access modifier, making them default, being changed to a declared access modifier.

The second category is the change to non-access modifiers. This means the changing, addition, or removal of Java keywords that do not modify access. Java non-access keywords include final, static, or void. While variable types are keywords, changing of variable types do not fall under this definition, as this type of keyword modification was not seen. In addition, the changing of loop types, such as for to whiles, may fall under this definition but needs more research. The most common type of this change is the addition of final variables. Figure 4.15 highlights this type of change, as the final keyword has been added.

Chapter 5

Analysis and Discussion

In this chapter, we take a closer look at the data and their relationship to the research questions. In Section 5.1, , we look at and discuss the performances of the targeted keyword patterns in more detail. In Section 5.2, we take a closer look at the taxonomy provided, examining prevalence of the categories, how they emerged, and what benefits they might provide to readability.

5.1 RQ1: What keyword patterns were most highly correlated with readability commits?

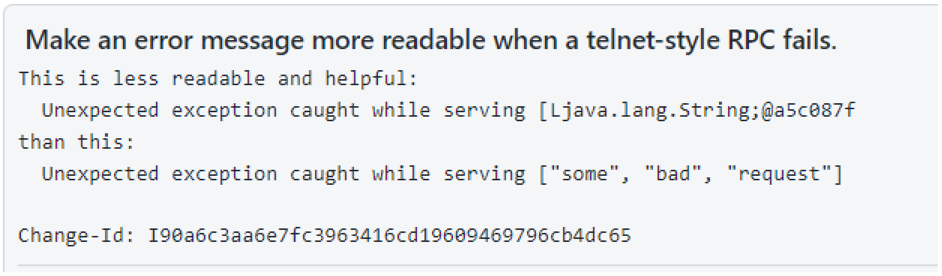
Table 5.1: Break Down of Keyword Occurrences

Keyword	**Total Found**	**Total Reviewed**	**Marked Readability Commit**	**Marked Non-Readability Commit**	**Precision**
'readab.'*	155	113	64	49	56.64%
'understand.'*	95	59	13	46	22.02%
'clean. up'*	1532	364	117	247	32.14%
Total	1782	536	194	342	36.19%

A key aspect to this project was the data collection. As mentioned in Chapter 3, the data set was partially collected via automatic collection and then manually reviewed. From the initial data set of refactoring commits, our tool automatically identified 1,782 potential readability commits based on targeted keyword patterns. Of these 1,782 commits, 536 commits would be manually reviewed. Table 5.1 shows the breakdown of the keywords patterns. Of the 536 commits looked at, only 194 of the commits were considered readability changes by the reviewer. This gives the tool a precision of 36.19%. This indicates generally that there is room for improvement with the tool.

While there are identifiable issues within individual patterns, the tool is also lacking in some respects that affect all patterns. The most common issue was the inability to differentiate test file changes from source code changes. Since test files were not considered in this study, the reviewer would simply mark them as non-readability commits. This brings down the precision of all the keywords slightly. There are a set of commits in which readability improvements occurred in test files and were called out by developers; however, reviewers marked them as non-readability commits. In a rework of this project, it would be better to simply throw out these types of commits all together. The other major issue was the reviewer's inability to identify a readability change. There were situations seen where the developer said a change occurred; however, the reviewer could not find the change. These would also be marked as non-readability commits.

The first keyword pattern selected for the search was *'readab.*'*. This pattern was meant to target specifically words like readability or readable. This was generally the best preforming pattern. Of the total 155 keywords identified, 113 were manually reviewed. Of these 64 commits were marked as readability, while 49 were marked as non-readability commits, giving a



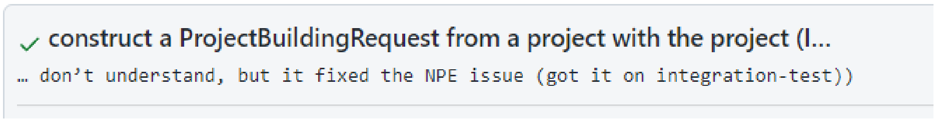
Make an error message more readable when a telnet-style RPC fails.

```
This is less readable and helpful:
  Unexpected exception caught while serving [Ljava.lang.String;@a5c087f
than this:
  Unexpected exception caught while serving ["some", "bad", "request"]

Change-Id: I90a6c3aa6e7fc3963416cd19609469796cb4dc65
```

Figure 5.1: Example of *'readab.*'* pattern considered a non-readability commit

precision of 56.64%. Some unique issues with the *'readab.*'* pattern were identified. The terms for this pattern were also used to describe the product of the software as opposed to the source code. This means that developers would often describe changes to log messages or UI elements as "more readable", which would be marked as a non-readability commit. Figure 5.1 shows an example of this, the "more readable" item is the error message itself, not the code that creates the message. This issue is caused by the simple nature of the mining tool. In addition, this pattern was small in its scale. From the larger database of commits from 800 software projects, only 155 commits were identified as belonging to this pattern. So, while one of the more precise patterns used, it also a rarely used pattern.

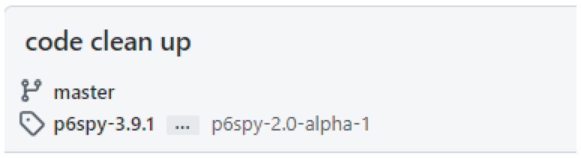


✓ construct a ProjectBuildingRequest from a project with the project (I...

```
… don't understand, but it fixed the NPE issue (got it on integration-test))
```

Figure 5.2: Example of *'understand.*'* pattern considered a non-readability commit

The second identified keyword pattern was *'understand.*'*. This pattern was chosen due to the linked nature of software readability and software understanding. This pattern's performance is the most surprising. Of the 95 commits identified with this pattern, 59 were reviewed and only 13 were identified as readability commits. This gives a precision of 22.02%. In the initial stages of the project, it was assumed that *'understand.*'* would be similar to *'readab.*'*; however, this was our worst preforming pattern in both terms of scope and precision. This pattern suffered from similar issues to *'readab.*'*, but a bit worse. It would also be used to describe the software's poducts as opposed to the source code, similar to *'readab.*'*, but would also have two other seen uses. Developers would also use this pattern to describe their own understanding of the code, either they understood or did not understand an aspect of the code or behavior. Figure 5.2 shows one of these commits, where the developer is actually describing their own understanding (or lack) of the code. Finally, the developers also used this pattern to describe the code's ability, like the ability to interpret file formats. Once again, these issues were created by the tools inability to understand the context of the patterns. This keyword was also very limited in its scope, being the least commonly detected pattern.


code clean up

master

p6spy-3.9.1 ... p6spy-2.0-alpha-1

Figure 5.3: Example of *'clean.* up'* pattern considered a non-readability commit

The final identified keyword pattern was *'clean.* up'*. This pattern was chosen to have a more generalized pattern that still focused on changing existing code. Since this was a more generalized term, it was by far the most common. Of the 1,535 commits with this pattern, 364 were reviewed and 117 were considered readability improvements. This gives the pattern a 32.14% precision. The main issue with this pattern type came from its vagueness. While there was indication that readability changes were made with these commits, most of these commits were not considered as readability changes. This was made worse by its usage, as developers would very often just call a commit “clean up” without context. It was difficult for the reviewer to find and determine readability changes without the context. These very generic commit messages would also often occur in commits with hundreds of line changes, making it very taxing to determine if a readability change occurred. Figure 5.3 shows one of these commits, where the developer simply state code clean up. While this keyword pattern represents most of the readability changes looked at, a rework of the pattern maybe needed to make it more accurate.

In addition to exploring the performance of our targeted patterns, we pulled word counts from all the readability commit messages to attempt to identify other usable patterns. After eliminating stop words, using NLKT’s stop words list, a count of all the words in the readability commit messages were taken. Table 5.2 shows the top 20 words discovered from 2,898 considered words. The first observation is the high variety in the words used for commit messages, as the most common word, code, only makes up 2.65% of all words. We also see the targeted patterns, such as clean, readability, readable, and cleaning which make sense as they were directly targeted. From this data, the most interesting words include added/add, better, fix/fixed, and removed. While not targeted by mining, these words occurred at a high rate

Table 5.2: Top 20 Detected Words

Word	**Count**	**Word**	**Count**
code	77	**readable**	16
clean	67	**class**	15
added	42	**improve**	15
cleaned	41	**add**	15
readability	28	**git-svn-id:**	14
removed	23	**fix**	14
method	21	**use**	13
better	20	**fixed**	13
make	19	**cleaning**	13
methods	17	**type**	12
Total			**2898**

and generally indicate a change to existing code bases. The issue is that we cannot link these words with readability as with our patterns. While addition was common, and some of our changes are additions (like comments), there is no direct way to tell if these added sections are readability changes. It must be acknowledged that while these words were in commits associated with readability changes, these words may not refer to readability changes at all. They may describe changes unrelated to readability change, as commits often involve a number of changes

5.2 RQ2: What code changes do developers make when they admit that they are trying to improve code readability?

5.2.1 Method Extraction

Table 5.3: Break Down of Method Extraction Occurrences

Category	Count	Precent of Total
Method Extraction	**64**	**20.38%**
If-else Extraction	15	3.18%
Boolean Logic Extraction	5	1.60%
Class Extraction	7	2.23%
Loop Extraction	5	1.60%
Try-Catch Extraction	5	1.60%

Looking first at the method extraction category, 64 total changes were associated with this category, representing 20.38% of examined changes. This makes this the single most common change discovered in the research. Since this was focused on software refactoring, it makes sense that method extraction would be well represented here as it is one of the more common software refactors. Of the subcategories: 15 were identified as if-else extraction, 7 were identified as class extraction, 5 were loop extraction and 5 were try-catch extraction. This highlights the very diverse nature of method extraction. Interestingly, the opposite of method extraction, inline method, was seen a couple times, but not enough to include it in the taxonomy.

Method extraction was one of the first emerging trends. It was identified very early on in the review process; however, some of the subcategories were harder to identify. As seen by the gap between method extraction, and the largest category of if-else extraction, method extraction is an operation that can be performed to a wide range of code. The trends of the code blocks did not appear until a re-review of all the existing method extractions were completed. This made it a little clearer that method extraction tended to be performed to control statements. At this time the distinction between if-else extraction and Boolean extraction was identified. Since the if's conditional statement is separated from the if-else statement, it emerged as its own category. As mentioned in the taxonomy section, class extraction is a curious inclusion here. Since we mostly identified methods moving to new classes, it was set as a subcategory of method extraction, but given more thorough research it could become its own category.

Overall, the prevalence of method extraction as a readability change coincides with some other research looked at. Fakhoury et al. identified the extract method refactoring as a common readability change. They identified extract methods occurring 124 times, categorized as a readability improvement [3]. In contrast, they saw many inline methods preformed, which was not seen in this research. In Piantadosi et al., they found that code refactoring improved software readability [11]. As method extraction is one of the most common refactors, it again makes sense that it would be a tool used by developers in readability improvements.

Method extraction makes sense as a common readability change for a few reasons. First, when performed on repeated code blocks, it simply reduces the amount of code the developer must read. Rather than attempting to read a block of code multiple times just to realize it is a reused segment, they can just reference the method. In addition, method extraction comes with the benefit of naming. Being able to associate a block of code with a descriptive method name can make that block of code easier to understand. It also gives the reader the option of abstracting out that segment, meaning they can ignore the code block if they feel it is irrelevant based on the name.

5.2.2 Identifier Renames

Table 5.4: Break Down of Identifier Rename Occurrences

Category	**Count**	**Precent of Total**
Identifier Rename	**57**	**18.15%**
Meaning Modified	27	8.60%
Meaning Preserved	16	5.10%

The second major category was the identifier rename category. Rename changes were performed 57 times, making up 18.15% of the changes looked at. Of those, 27 were identified as renames that modified the meaning of the identifier, while 16 preserved the identifier's meaning. Not all the identified renames fit within one of the two subcategories; even though they are dichotomies. Since this methodology treated all renames in a commit as one singular change, the reviewer did not specify the renames which occurred, just stating they did occur. This most often occurred when many different renames were performed in a commit. This also highlights another observation with the renames. They often occur in batches, with several renames occurring in a single commit. If each rename in a commit was categorized as a unique change as opposed to a batch, rename operations likely become the most seen change.

Identifier renames was another category that was identified right away in the process. As with most of the larger categories, the first few changes seen in the review involved a rename in some way. As the breakdown of the category occurred, there were two potential ways to go with it, either based on what is being renamed or how the rename was done. The first category more closely follows the taxonomy presented by Fowler, who distinguished rename types on what is being renamed [16]. Our review only noted two software entities that tended to be renamed, methods and variables. To focus more on the readability aspect, the effects on the identifiers meaning was looked at more closely. This led to the division of "is the meaning modified or preserved". While these criteria were mentioned in the taxonomy, these were general guidelines based on the reviewer's experience.

In Arnaoudova et al. [17], a similar breakdown of rename operations is seen. In their REPENT taxonomy, they provide an in-depth summary for categorizing rename operations [17]. In this taxonomy, one key category is semantic changes. This category is very similar to the trend that we observed with our renaming. Arnaoudova et al. divided these into 7 categories including, preserved meaning, changed meaning, narrow meaning, and broaden meaning [17]. While our taxonomy does focus on semantic changes, it is not as branched as REPENT, considering all modifications in meaning as a change in meaning. The REPENT taxonomy is also more significant in scope, categorizing forms and grammatical changes, which were not seen within our observations [17]. Ultimately, the REPENT taxonomy supports the existence of the observations we made for our rename taxonomy, as similar focus on semantic changes were seen.

Identifier renames were a change that was expected to be the most common. This is because of the role that identifiers play within software. Since they have looser syntax rules, they are one of the best opportunities to provide the reader with information. The importance of identifier quality is something that is taught to every CS1 student. Naturally, changing poorly worded or incorrect identifiers to more accurate ones fix potential readability issues that occur. Again, the research in Fakhoury et al. also identified rename refactoring necessary readability change. A significant portion of the refactors they looked at which were associated with readability was rename refactoring [3]. Martin's book, *Clean Code: A Handbook of Agile Software Craftsmanship* also addresses the importance of identifier naming, highlighting the importance of meaningful names [2]

5.2.3 Code Reformatting

Edits to the code format was the third most common category. Of the changes, 34 of them were of this type representing 10.8% of the changes looked at. Interestingly, this category was dominated by changes to white spaces, as 32 of the changes contained some type of white space edits, roughly 10.19% of

Table 5.5: Break Down of Code Reformatting Occurrences

Category		Count	Precent of Total
Code Reformatting		**34**	**10.80%**
	Changes to Code Whitespace	32	10.19%
	Changes to Brackets	5	1.60%

the changes seen. These changes to white space tended to be massive scale changes, reformatting entire files to unify indexing or spacing. Less common was the changes to bracketing, seeing only 5 changes to brackets. These were almost always done with white space changes. The only type seen that did not include a white space change was the addition of curly braces around one line if statements, which only occurred twice on its own.

Code reformatting was another one of the earliest identified categories. During the research, method extraction, renames, and reformatting made up what was the big three. Those three changes were seen as the most obvious and apparent changes. Code reformatting was almost entirely done using white space edits, either spaces, new lines, or indentations. This change can be traced back to some of the earliest research into software readability models. Buse and Weimer's initial software readability metrics focused on white space, specifically indentations and blank lines [5]. Martin also calls out the importance of code formatting, mentioning the importance of vertical and horizontal formatting [2]. The more surprising thing was the lack of edits to braces, such as parentheses or curly braces were modified, likely due to language syntax.

The formatting of software also provides an opportunity to improve the readability of software code bases. In Java specifically, the formatting of the code is more fluid, not requiring indentation, bracket position or white space.

Languages like Python, which was developed with a focus on readable syntax, enforce white space requirements. This makes code reformatting, and specifically white space changes, more useful based on language. The use of blank lines to separate code blocks can be done in both Java and Python; however, indentation is a Python requirement.

5.2.4 Code Removal

Table 5.6: Break Down of Code Removal Occurrences

Category	**Count**	**Precent of Total**
Code Removal	**33**	**10.51%**
Removal of Dead Code	21	6.69%
Removal of Redundant Code	12	3.82%

The fourth most common change observed was the removal of code. 33 of the changes observed, 10.51% of the total, involved the deletion of code. Of these, 21 were identified as removal of dead code, while 12 were the removal of redundant code. The main distinction between the dead code and redundant code is if the code is used, as dead code is generally considered to be code that is not being used. This distinction is obvious in most cases, as commented out or never called code is clearly dead code. The difference between redundant code and dead code which was executed was left up to the reviewer. Code removal was discovered in a bottom-up manner, meaning the subcategories were identified and the link was discovered later in the review. This is unique from the other three methods explored, which were discovered in a top-down manner. In addition, code removal represents the first unique trend discovered in this research, as other similar research did not directly identify code removal as a readability change. While the code removal was mostly done with clean-up operations, there is some link to the software's readability. Like method extraction, it can reduce the amount a developer needs to read. In addition, it can help reduce confusion. Dead code can be inherently confusing, as the assumption is that the code does something or has some effect. If the code does not influence the system, this breaks that assumption potentially confusing the reader. By cleaning out code segments which are not useful or used, you lessen the strain on the developer reading the code.

5.2.5 Addition of Comments

One of the smaller categories of the taxonomy was the Addition of Comments. Of the changes looked at, the addition of comments occurred 23 times. This category is unique in that it could not be broken into subcategories. There were no real observations made in which this class could be further broken down. Comments were only really added as Javadoc, making only one distinct category. While not all comment adding commits were Javadoc, no real trend among the comments could be seen. Making it more difficult to distinguish categories is that when general comments were added so where Javadocs, so reviewers often tagged comments and Javadocs together. In Pascarella and Bacchelli, a more thorough taxonomy of comments is presented [18]. This taxonomy focused on identifying different purposes, locations, and styles; however, their research did not tie comments to readability improvements. Finally, Martin includes comments and their importance to readable code in a chapter of his book [2].

The addition of comments is a debatable readability improvement. While it is true that comments increase code understanding, they are often viewed as supplemental to the code itself. Our research considered any change to the java files, so while external documentation would not be included, comments were included as acceptable changes. Since comments exist within the code and help improve the general code understanding for readers, they are considered a readability change.

5.2.6 Keyword Modification

Category	**Count**	**Precent of Total**
Keyword Modification	**14**	**4.46%**
Non-access Keyword	8	2.55%
Access Keyword	6	1.91%

The final category from the taxonomy was the modification of keywords. This represented only 14 changes, about 4.46% of the changes looked at. This, like code removal, was a group created as a parent to two emerging trends. The first and more common of the two was the modification of non-access modifiers. 8 times we saw direct changes to non-access modifiers, most often the addition of the final or static keywords. The less common trend was the modification of access keywords. Of the 6 changes categorized here, the most common change seen was the transition from a non-declared access, so no keyword, to a declared access. This was another change that was not identified in other research looked at.

As for this category as a readability change, it is again a debatable one. More than any other category, this category has the most direct effect on the actual behavior of the program. A switch from private to public, or the final addition, has a programmatic effect and changes some of the behavior. On the other hand, an intrinsic amount of information is provided by keywords, especially with the addition of optional keywords. Since these changes were seen within targeted rename commits, ultimately it was included in the taxonomy.

Chapter 6

Related Work

Several studies have looked at how to improve software maintenance in general [19–98]. Among them, we focus, in this section, on studies (1) measuring readability in code, (2) detecting readability improvement opportunities, and finally (2) recommending readability-related refactorings.

6.1 Research of Software Readability Models

Software readability models represent some of the earliest attempts at quantifying software readability and tracking readability improvements. Readability models can be tracked back to Buse and Weimer's work into creations of various software readability metrics [5]. The Buse and Weimer models focused more on the key statistics of the code, looking at things like line length or identifier length [5]. These models would be expanded upon by Dorn [6] and Scalabrino et al. [8]. Their work would make these models more advance. Scalabrino et al. would go on to extend their readability model, including a focus textual features [9].

Roy et al. [7]. would identify some issues within the existing models, arguing that they used more statistical approaches which miss important aspects of readability. Roy et al. used a very similar method of identifying readability commits as we used. They mined keywords from Java commits, including words which match our *"readab.*"*, *"under.*"* and *"clean.* up"* patterns. These commits would then be manually reviewed to ensure they were readability commits. The distinction is that these commits would be used to create a dataset, which would have metrics taken from it and used to create a new software model.

6.2 Research of Software Readability Improvements

Robert Martin's book *Clean Code: A Handbook of Agile Software Craftsmanship* [2] conveys a large amount of information from professional developers on the creation of "clean code". A large amount of the book focuses on how to create extremely readable code There is a large amount of overlap between Martin's recommendations in his book and the changes found within this study. Martin's focus is more on creating new code, rather than the improvement of existing code; however, his advice is still applicable. Many of the observations made in this research are also found within Martin's book, further strengthening the trends we have found.

Fakhoury et al's [3] explored the different types of changes made by developers to improve readability. The goal was to test the ability of readability models to detect incremental changes. Again, they used a similar method of detection of readability commits as this research, using keyword mining of commits to detect self-admitted readability changes. While these commits would be used to test the model's effectiveness, Fakhoury et al. also explored some of the changes developers made. These commits were analyzed using source code analysis tools, like *RefactoringMiner* and *CheckStyle*. Similar trends to those found in this paper were made, such as using method extraction as a common readability change; however, the automation may have missed other trends. While some automation was used in the detection of commits, manual review was performed for the identification of trends. In addition, they explored how developers document their readability commits, which is a similar contribution to this paper's RQ1.

Piantadosi et al [11] preformed research in how the readability of software changed generally over time. Their research found that the developers tend to have readability in mind from the start and that readability changes very little over time. More relevant to this research, they also explored a bit of the changes made to files that had significant readability changes, determined by their models. In this research, one of their major conclusions was that refactors had major impacts on code readability; however, these ties are not explicitly tied to readability improvement attempts by developers.

6.3 Research of Renaming

Arnaoudova et al. [17] developed a more complete taxonomy of rename changes, known as REPENT. Within their taxonomy, they presented different aspects of identifier renames including location, semantic changes, rename forms and grammatical changes. Most notably, the semantic change mirrors our choices for the rename taxonomy. Our breakup of readability was based on semantic change, looking at if the meaning changed or was preserved. The REPENT taxonomy included these categories and included aspects like name broadening, narrowing, and adding or removing meaning. All these additional aspects were simply considered a modified meaning in our taxonomy. Our taxonomy is further supported and potentially expanded upon with the REPENT taxonomy; however, our research solidifies the connection to readability.

Pernuma et al. 2018 [99] further expanded upon their existing research into software renames. In their study, they explored more closely how developers rename and what is the motivation. They used the REPENT taxonomy from Arnaoudova et al. to explore what types of renames developers had performed. This included looking at if the meaning of identifiers were modified or preserved, with most identifiers being modified. This lines up with some of the observations found within this research, as we found in a much smaller sample that meaning tended to be modified by developers.

6.4 Research of Refactoring

Several works related to the effects refactoring has on software. One of the primary sources for refactoring is Martin Fowler's book *Refactoring: Improving the Design of Existing Code* [16]. Fowler's book is one of the most complete works on taxonomizing refactors. Since our work focused on refactors, specifically those from *RefactorMiner*, many of the Fowler refactors were seen. As noted in Chapter 4, a large amount of our taxonomy is made up of refactors from Fowler's taxonomy.

AlOmar et al. 2021 [100] took preformed research into self-admitted refactors by developers. Similarly, to how we determined self-admitted readability improvements by developers, AlOmar et al used commit messages to determine if a change occurred. These commits were then used to develop a machine learning algorithm to detect and classify refactors. The most similar aspect of their research is the aspect of self-admitted refactors making up their dataset; however, they focused on the refactoring research.

6.5 Other

Pascarella and Bacchelli [18] created a taxonomy of comments in Java projects. By performing an extensive manual review of software comments, they developed a thorough taxonomy of the types of comments developers add to source code. Their research into comment taxonomy could be used to expand upon the taxonomy of comments; however, their research did not focus on readability, but instead comments.

Chapter 7

Threats to Validity

While we present experimental results based on real life projects, identified factors within the study impact the applicability of the observations made. These threats to validity have been divided into four categories [101].

Conclusion Validity. Threats to conclusion validity focus on issues with the ability to draw the correct conclusions from the relations between the observation and the outcome [101]. The main threat to this type of validity comes from the manual review process. In some cases, the reviewer had to make a subjective decision if a readability change occurred. This most often occurred with very general and unspecific commit messages. This subjective decision may have led to some bias in what observations were considered. In addition, if the reviewer determined the change was a readability change, that also introduces some subjectivity into the observations. These additions of more subjective decisions may affect the conclusion made. This was mitigated by mostly focusing on changes that were properly described by commit messages and criteria for what constituted a readability change. Still, subjectivity and bias were likely introduced by the reviewer.

Internal Validity. Threats to internal validity focus on influences on the independent variable that could not be controlled for [101]. The most apparent threat to internal validity comes from the review sessions' nature. Since the manual review of software changes can take time, the review was done via several sessions. These sessions did not have consistent timing, meaning that some sessions were quick while others were longer. These longer sessions could become taxing on the reviewer, leading to less accurate observations over the time of this session. This issue is exacerbated by the inaccuracy of the initial sampling tool. Since the reviewer also needed to determine if a readability change occurred at all, more review was done leading to even longer more taxing sessions. This could lead to inconsistent observation quality or even missed observations, which affect the presented research.

Construction Validity. Threats to construction validity are issues with generalization to the concept or theory of the study [101]. The main threat to this validity comes from our lack of resources. One person performed all the reviewing, only reviewed after the fact by others. This means that this one reviewer's bias is inherent to the observations made. In addition, this same reviewer did most of the memoing. This again introduces bias to the observations. The repeated nature of the memoing also has another effect. It is possible that the reviewer made observations in trends and was biased in seeing them. This would help in the identification of these trends; however, it might blind them to new trends. If this study was to be repeated, having more reviewers and division between commit reviewers and observation reviewers may mitigate these biases.

External Validity. Threats to external validity limit the ability to generalize the results of the experiment to industrial practice [101]. The main threat to external validity comes from the use of only Java projects. While Java was the focus because of common usage, these results can only truly represent trends in Java. It is unknown how generalizable these observations are to other languages, though the study does use a statistically significant sample of Java projects; mitigating the threat that the results presented do not generalize to open source Java systems. It is possible many of these changes are preformed specifically because of Java's language features. An example was already pointed out in the use of indentation. While this change is possible in Java, Python's white space requirements make it unlikely to see this type of change. In addition, the focus on specifically exploring refactoring commits may limit the generalizability. It is possible that readability is improved outside commits where refactoring is performed, and those changes are maybe different. While the focus on refactoring commits helped focus on improvements, it also limits the observations to refactor commits. In addition, it does bias some of the observations to refactors instead of other possible readability changes.

Chapter 8

Future Work

In the future it is hoped that this research is expanded in scope. First, with a more substantial review to help support the trends identified, while eliminating some shortcomings. In addition, an expansion to additional languages may highlight different trends or differences in approach. These are both done in a hope to make a much more substantial and supported taxonomy. In addition, improvements could be made to the identification of readability commits presented in this research. More accurate and effective methods of detecting commits where the developers wish to improve readability could improve the taxonomy's validity. Exploring keyword patterns and committing message trends could improve the automatic methods' effectiveness. Finally, this research only looked at readability commits and improvements; however, different trends may exist in non-refactoring commits. All this work would improve the overall taxonomy and its generalizability.

Chapter 9

Conclusion

The objective of this work was to gain a deeper understanding of the types of changes developers make when attempting to improve the readability of their software. To achieve this, an automatic process using keyword pattern mining was developed to search commits for indications of readability improvements. This mining was applied to a set of refactoring commits from 800 open-source java projects. This created a data set of 1,782 potential readability commits, of which 536 commits would be manually reviewed. From this process, several takeaways have been provided.

9.1 Takeaways from RQ1

The keyword pattern *readab. provides the best accuracy for detecting readability improving commits; however, it has a limited usage**. When looking closely at the keyword patterns used, our *readab.** pattern represented the highest precision in the detection of readability commits.

While a 56.64% precision is not highly precise, our detection method was very naive and generally lacked precision. The limiting factor is that this pattern was only seen 155 times, meaning its usage was limited. This limited usage makes it a bit unreliable for the future detection.

The *understand. keyword pattern was the worst keyword pattern for readability improvement detection.** This keyword pattern had the worst precision and smallest scale. The precision of the *understand.** keyword pattern was only 22.02%, meaning that only around 1/5 of the looked at commits were tied to readability by reviewers. In addition, this keyword was only detected 95 times, further limiting the scope of the pattern.

The *clean.* up* keyword pattern, while very common, is too general to reliably detect readability commits. The bulk of detected keyword patterns in the sample data set followed this pattern, with a total of 1,532 commits matching this pattern. The issue is that this pattern is far too general and very loosely linked with specifically readability improvements, as the sample of 364 commits only gave a precision of 32.14%. This keyword pattern was mostly harmed by poor commit messaging, as it was common for the looked at commits to just be "code clean up".

It is difficult to link keyword patterns to direct readability change given our approach. Our approach only gave us a 36.10% precision, which is very inaccurate. The issue comes partially from the process, where commits that maybe should have been disregarded were marked as non-readability. Some of it comes from the patterns chosen, as *understand.** and *clean.* up* proved to be inaccurate. Finally, sometimes commit messages just are not descriptive.

9.2 Takeaways from RQ2

ṡec:conRQ2

Method extraction was the most seen readability improvement. Seen 64 times in the data set, the removal of code to a new function was the most seen change made by developers when attempting to improve readability.

Developers often preformed Identifier Renames when attempting to preform readability improvements. Although, 57 of the looked at changes involved some sort of rename operation, generally multiple renames would occur.

Developers would often reformat the code base to improve its readability. 34 of the changes looked at involved changes to code formatting, the majority of which affected the code's whitespace. This change often occurred within clean up commits.

Developers would often remove dead or unused code when improving readability. This change was seen in 33 of the explored operations. Again, this change was seen mostly as part of clean up operations

Comments can be added to improve the source code readability. 23 of the changes looked at included the addition of comments, mainly the addition of Javadoc.

Changes to reserved keywords can potentially improve readability. 14 of the changes looked at involved changes to software keywords.

Overall, this research presents a taxonomy of software readability changes. This taxonomy provides several contributions to both research and industry. First, this taxonomy can be used to further explore and understand readability improvements, enforcing existing research and presenting new trends. In addition, it provides several changes developers could make if wanting to improve their own code. Finally, the taxonomy can be used to support documentation, giving names to change types as seen with other taxonomies, like Fowler's Refactorings [16]

Bibliography

[1] Xin Xia, Lingfeng Bao, David Lo, Zhenchang Xing, Ahmed E. Hassan, and Shanping Li. Measuring program comprehension: A large-scale field study with professionals. *IEEE Transactions on Software Engineering*, 44(10):951–976, 2018.

[2] Robert Martin and an O'Reilly Media Company Safari. *Clean Code: A Handbook of Agile Software Craftsmanship*. Pearson, 1st edition, 2008.

[3] Sarah Fakhoury, Devjeet Roy, Adnan Hassan, and Vernera Arnaoudova. Improving source code readability: Theory and practice. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, pages 2–12, 2019.

[4] Sarah Fakhoury, Yuzhan Ma, Venera Arnaoudova, and Olusola Adesope. The effect of poor source code lexicon and readability on developers' cognitive load. ICPC '18, page 286–296, New York, NY, USA, 2018. Association for Computing Machinery.

[5] Raymond P. L. Buse and Westley Weimer. A metric for software readability. In *ISSTA '08*, 2008.

[6] Jonathan Dorn. A general software readability model. 2012.

[7] Devjeet Roy, Sarah Fakhoury, John Lee, and Venera Arnaoudova. A model to detect readability improvements in incremental changes. In *Proceedings of the 28th International Conference on Program Comprehension*, pages 25–36, 2020.

[8] Simone Scalabrino, Mario Linares-Vásquez, Denys Poshyvanyk, and Rocco Oliveto. Improving code readability models with textual features. In *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, pages 1–10, 2016.

[9] Simone Scalabrino, Mario Linares-Vásquez, Rocco Oliveto, and Denys Poshyvanyk. A comprehensive model for code readability. *Journal of software : evolution and process*, 30(6):e1958–n/a, 2018.

[10] Jevgenija Pantiuchina, Michele Lanza, and Gabriele Bavota. Improving code: The (mis) perception of quality metrics. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 80–91, 2018.

[11] Valentina Piantadosi, Fabiana Fierro, Simone Scalabrino, Alexander Serebrenik, and Rocco Oliveto. How does code readability change during software evolution? *Empirical software engineering : an international journal*, 25(6):5374–5412, 2020.

[12] Klaas-Jan Stol, Paul Ralph, and Brian Fitzgerald. Grounded theory in software engineering research: A critical review and guidelines. 05 2016.

[13] 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, apr 2021.

[14] Anthony Peruma, Mohamed Wiem Mkaouer, Michael Decker, and Christian Newman. Contextualizing rename decisions using refactorings and commit messages. 08 2019.

[15] Nuthan Munaiah, Steven Kroh, Craig Cabrey, and Meiyappan Nagappan. Curating github for engineered software projects. *Empirical Softw. Engg.*, 22(6):3219–3253, dec 2017.

[16] Martin Fowler and an O'Reilly Media Company Safari. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley Professional, 1st edition, 2018.

[17] Venera Arnaoudova, Laleh M. Eshkevari, Massimiliano Di Penta, Rocco Oliveto, Giuliano Antoniol, and Yann-Gaël Guéhéneuc. Repent: Analyzing the nature of identifier renamings. *IEEE Transactions on Software Engineering*, 40(5):502–532, 2014.

[18] Luca Pascarella and Alberto Bacchelli. Classifying code comments in java open-source software systems. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 227–237, 2017.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] Hussein Alrubaye, Stephanie Ludi, and Mohamed Wiem Mkaouer. Comparison of block-based and hybrid-based environments in transferring programming skills to text-based environments. *arXiv preprint arXiv:1906.03060*, 2019.

[33] 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.

[34] Christian D Newman, Mohamed Wiem Mkaouer, Michael L Collard, and Jonathan I Maletic. A study on developer perception of transfor-

mation languages for refactoring. In *Proceedings of the 2nd International Workshop on Refactoring*, pages 34–41, 2018.

[35] 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.

[36] 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.

[37] 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.

[38] 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.

[39] 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.

[40] 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.

[41] 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.

[42] 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.

[43] 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.

[44] 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.

[45] 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.

[46] 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.

[47] 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.

[48] 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.

[49] 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.

[50] 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.

[51] 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.

[52] 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.

[53] 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.

[54] 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.

[55] 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.

[56] 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.

[57] 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.

[58] 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.

[59] 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.

[60] 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.

[61] 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.

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

[63] 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.

[64] 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.

[65] 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.

[66] 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.

[67] 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.

[68] 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.

[69] 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.

[70] 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.

[71] 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.

[72] 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.

[73] 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.

[74] 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.

[75] 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.

[76] 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.

[77] 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.

[78] 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.

[79] 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.

[80] 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.

[81] 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.

[82] 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.

[83] 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.

[84] 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.

[85] 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.

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

[87] 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.

[88] 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.

[89] 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.

[90] 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.

[91] 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.

[92] 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.

[93] 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.

[94] 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.

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

[96] 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.

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

[98] 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.

[99] Anthony Peruma, Mohamed Wiem Mkaouer, Michael Decker, and Christian Newman. An empirical investigation of how and why developers rename identifiers. pages 26–33, 09 2018.

[100] 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.

[101] Claes Wohlin. *Experimentation in software engineering*. Springer, New York;Berlin;, 2012.