# What Changes in Where? An Empirical Study of Bug-Fixing Change Patterns

Md Rakibul Islam
University of Wisconsin - Eau Claire
Eau Claire, Wisconsin, USA
islamm@uwec.edu

Minhaz F. Zibran
University of New Orleans
New Orleans, Louisiana, USA
zibran@cs.uno.edu

## ABSTRACT

A deep understanding of the common patterns of bug-fixing changes is useful in several ways: (a) such knowledge can help developers in proactively avoiding coding patterns that lead to bugs and (b) bug-fixing patterns are exploited in devising techniques for automatic bug localization and program repair.

This work includes an in-depth quantitative and qualitative analysis over 4,653 buggy revisions of five software systems. Our study identifies 38 bug-fixing *edit* patterns and discovers 37 new patterns of nested code structures, which frequently host the bug-fixing edits. While some of the *edit* patterns were reported in earlier studies, these *nesting* patterns are *new* and were never targeted before.

## CCS Concepts

•**Software and its engineering** → **Maintaining software; Empirical software validation;** *Correctness; Error handling and recovery;* **Software defect analysis;** *Correctness; Software reliability; Error handling and recovery;* **Software defect analysis;** *Empirical software validation; Maintaining software;*

## Keywords

Software; Defect; Bug; Fault; Error; Vulnerability; Source Code; Edits; Nesting; Pattern; Empirical Study; Analysis

## 1. INTRODUCTION

Technology today rarely exists without a software component or interface as more and more systems are being software operated. But, incidents of software failures and vulnerabilities repeatedly make news headlines since the emergence of software to date. In 2017 alone, almost *$1.7 trillion* in assets, and *3.7 billion* people were affected by software failures [22].

These software failures occur primarily due to software bugs or defects (a subset of which are exploitable security vulnerabilities). There are mainly two mutually complementary approaches to minimize software bugs. First, the software developers must proactively adopt defensive programming practices to avoid common coding patterns, which result in software defects. Second, (semi-)automated techniques for the detection and refactoring of bug-prone code patterns can help in program sanitization for minimizing software bugs.

Copyright is held by the authors. This work is based on an earlier work: SAC'20 Proceedings of the 2020 ACM Symposium on Applied Computing, Copyright 2020 ACM 978-1-4503-6866-7. http://dx.doi.org/10.1145/3341105.3373880

Till date, bug-fixing activities heavily depend on human efforts and consume a vast amount of total expenses in software maintenance [14] while nearly 80% of software cost is spent in maintenance [27]. Typical bug-fixing efforts mainly involve two types of tasks: (a) bug localization and (b) bug-fixing edits.

Bug localization deals with identifying the locations of the faulty pieces of code that cause the program to malfunction or remain vulnerable. Bug-fixing edits include subtle changes to program's source code elements, such as replacement of literals and insertion of conditionals, to eliminate the defect. Thus, a bug-fixing change encompasses two things: the edits made to the program elements and the location of those edits in the program's nested code hierarchy. Therefore, with respect to bug-fixing changes, we define the following two terminologies:

**Bug-fixing *edit* patterns,** which include the most frequent edit operations made to program elements for fixing a bug.

**Bug-fixing *nesting* patterns,** which include the most common locations in source code characterized as the nesting levels of code constructs that frequently host bug-fixing edits.

As bug-fixing changes indicate the locations of the buggy code and the original program elements found defective, a deep understanding of the common bug-fixing patterns can immensely help in minimizing efforts in both of the aforementioned two types of tasks (i.e., bug localization and edits) and can also contribute to devising techniques for automated program repair. A good understanding of the bug patterns can also help a developer to proactively avoid writing code that leads to program faults.

Bug-fixing efforts require a good understanding of the source code, intended edits, and their potential impacts. Studies [40, 42] find that code changes are repetitive in nature within and across code bases. Hence, mining code changes has become an effective way for program comprehension and deriving patterns of diverse categories including bug-fix patterns.

Early efforts in discovering bug-fix patterns highly depended on manual efforts [40, 57] in the analysis of textual differences among different program entities. However, manual effort is criticized for being error-prone, tedious, incomplete, and imprecise [17, 26, 45]. Recent efforts made use of Abstract Syntax Tree (AST) based code differencing tools (e.g., `ChangeDistiller` [19], `Diff/TS` [25] and `GumTree` [17]) for automatic discovery of code-changes and differencing program entities.

Previous work on discovering bug-fix patterns remained focused on bug-fixing *edit* patterns, which include bug-fixing changes to source code at a very fine-grained level without capturing those changes' surrounding code contexts such as *nested code structures.* The nested code structure, which is a hierarchy of AST nodes, indicates the location of a bug-fix change in an AST nodes' hierarchy. Nested code structures provide an important code context/aspect of bug-fix changes but remained absent in the studies [17, 38, 45, 46, 47, 57, 63, 69] that identified bug-fixing edit patterns.

In this work, we capture both bug-fixing *edit* patterns and *nesting* patterns (i.e., frequent nested code structures) of bug-fixing edits through an in-depth (quantitative and qualitative) analysis of 4,653 buggy revisions of five software systems drawn from diverse application domains. We organize this paper around two research questions as follows:

**RQ1**: *What are the common patterns of bug-fixing edits*?

– Here, we explore the subtle edits/changes made in source code for fixing bugs and we identify the bug-fixing *edit* patterns. We will verify what portion of the identified bug-fixing *edit* patterns are new, and how many of them were previously reported in earlier studies [47, 57, 63].

**RQ2**: *What are the prominent nested code structures that frequently host bug-fixing edits*?

– Here we investigate the frequent *nested code structures* (i.e., nesting patterns) where the bug-fixing edits are commonly located. These *nesting* patterns will directly contribute to bug localization, and will complement the *edit* patterns in our understanding of bug-fixing patterns with information about the locations and contexts of individual edits within surrounded nested code structures. Moreover, such AST based nested code structure contexts provide a potential scope to use those along with other code contexts such as *textual similarity of code* [62] to develop an effective technique to automatically locate program faults and repair those.

**Contributions:** Towards a deeper understanding of bug-fix patterns, this paper makes two major contributions:

- We identify a total of 38 bug-fix patterns organized in 14 categories. This is the highest number of bug-fix patterns identified in a single study. Four of these patterns are completely new, and the rest 34 edit patterns confirm those reported in earlier studies.
- We study locations of bug-fix changes in nested code structures and identify 37 *nesting* patterns (organized in six categories) that hold the majority of the bug-fixing edits. These *nesting* patterns are new (i.e., never targeted before), and add a new dimension in our understanding of bug-fix patterns.

This paper is an extension of our recently published work [33]. Especially, we have included here a deeper analysis, more detailed findings with additional results, as well as a more elaborated discussion of the methodology and results.

The remainder of this paper is organized as follows. In Section 2, we describe the methodology of this study including short descriptions of the dataset and tools used. In Section 3, we identify dominant bug-fixing *edit* patterns and answer RQ1. To answer RQ2, in Section 3, we derive the nesting patterns by distinguishing those nested code structures that host frequent bug-fix edits. In Section 5, we describe possible limitations and the threats to the validity of this work. Related work is discussed in Section 6. Finally, Section 7 concludes the paper with future research directions.

# 2. METHODOLOGY

The procedural steps of our empirical study are summarized in Figure 1. For each subject system, we collect the bug-fixing revisions. Then, for each bug-fixing revision, using AST based code differencing tools, we detect differences between the bug-fixing revision and its immediate previous revision. Collections of such AST differences are then analyzed to detect bug-fixing edit patterns and dominant nested code structures of code changes to fix bugs. In the following, we describe the subject systems and elaborate the procedural steps with necessary details.

## 2.1 Subject Systems

We study 4,653 revisions of five open-source software systems written in Java. These subject systems, as listed in Table 1, are available at GitHub. In Table 1, we present the total number of revisions and the number of source lines of code in the last revision. Here, KLOC denotes a thousand lines of code. We choose these five subject systems as these systems have variations in application domains, sizes, number of revisions, and are also used in other studies [31, 59].

Moreover, the selected five subject systems can be classified into two sets: (i) the first three subject systems, which were never been used earlier to detect bug-fixing *edit* patterns, belong to the first set, and (ii) the second set consists of the remaining two subject systems, which were earlier used in other studies [62]. Such a combination of selected subject systems provides the opportunity not only to verify the existence of previously reported bug-fixing edit patterns but also to identify new bug-fixing edit patterns if they exist in our dataset.

## 2.2 Distinguishing Bug-fixing Commits

For the top three systems in Table 1, we collect the bug-fixing commits identified by Ray et al. [59]. These three systems (i.e., Netty, Presto, and Facebook SDK for Android) were also used in other studies [31, 32].

The method for distinguishing the bug-fixing commits for a project/system is as follows. The commit messages associated with each commit during development of a project were

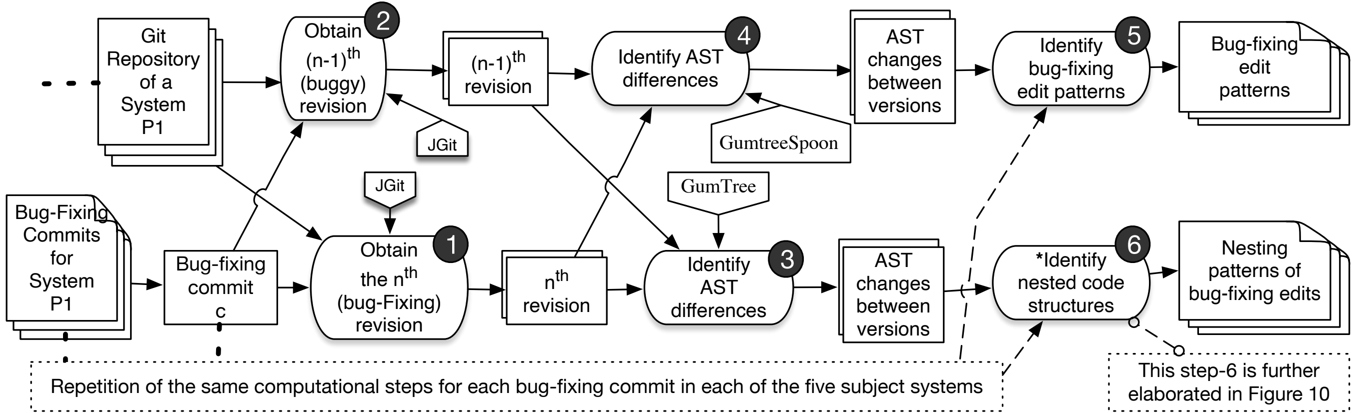



**Figure 1: Procedural steps (enumerated) to identify *edit* patterns and *nesting* patterns of bug-fixing changes**

**Table 1: Subject systems used in this work**

| Subject System | Short Description (Application Domain) | KLOC (last rev.) | Total # of Revisions | # of Bug-Fixing Rev. |
|---|---|---|---|---|
| Netty [6] | Event-driven network appication framework | 1,078 | 8,534 | 1,103 |
| Presto [8] | Distributed SQL query engine for big data | 2,869 | 11,909 | 841 |
| Facebook SDK for Android [3] | Social networking app. development framework | 172 | 671 | 133 |
| Apache Accumulo [1] | Distributed key-value store | 458 | 9,734 | 1,941 |
| Apache Common-maths [2] | Math library of math and statistics components | 187 | 6,971 | 635 |
| **Total over all the systems** | | 4,764 | 37,819 | 4,653 |

analyzed using standard natural language processing techniques. Each commit message was first converted to a bag of words, which was then stemmed. A commit is characterized as a bug-fixing commit, if the corresponding stemmed bag of words include one or more of the following keywords: 'bug', 'defect', 'fault', 'flaw', 'error', 'mistake', 'incorrect', 'issue', 'fix', and 'type'.

This approach for distinguishing bug-fixing commits was used in research [59, 50] and was reported 96% accurate [59]. To identify the bug-fixing commits in the remaining last two systems, we use the same keywords and approach as described above. The number of bug-fixing revisions for each system is listed in the right-most column of Table 1.

## 2.3 Computing AST-Differences

Consider a bug-fixing commit $\mathcal{C}$ resulting in the $n^{\text{th}}$ revision of a system/project $\mathcal{P}_1$. If a particular line of code $\mathcal{L}$ is modified in the bug-fixing commit $\mathcal{C}$, then it implies that the modification is necessary to fix the bug. Thus, the line of code $\mathcal{L}$ in the $(n-1)^{\text{th}}$ revision is considered a buggy line. In other words, we consider the changes between the $n^{\text{th}}$ and $(n-1)^{\text{th}}$ revisions of $\mathcal{P}_1$ are buggy. Several other studies [28, 29, 52, 58] also adopted the similar approach for distinguishing buggy source code.

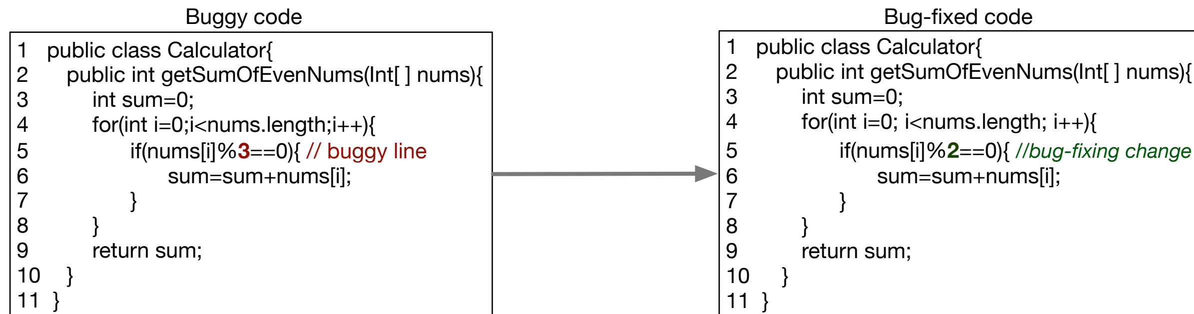
At this level, as shown in Figure 1, we obtain the $n^{\text{th}}$ and $(n-1)^{\text{th}}$ revisions using `JGit` [5]. Then, we capture bug-fixing changes at the AST [17] level between those two revisions using `GumtreeSpoon` and `Gumtree` separately (see action 03 and 04 in Figure 1). Captured AST differences using `GumtreeSpoon` and `Gumtree` are further processed to determine bug-fixing edit patterns and nesting patterns, respectively (see action 05 and 06 in Figure 1).

Before describing how we identify bug-fixing edit patterns (in Section 3) and nesting patterns (in Section 4), in the following, we discuss and compare the outputs of `GumTree` and `GumtreeSpoon` to develop background/context that helps in understanding the rest of the content of the paper.

### 2.3.1 *Interpreting GumTree's output*

For each action/change in a node, `GumTree` generates four major attributes: (i) *action name* (e.g., ins, del, upd or, mov) (ii) *label*- that indicates text/name of the changed node (iii) type of the changed node (e.g., changed node can be a simple `variable name` or an `expression`) and (iv) *nested code structure (NCS)*- the tree/hierarchy of parent nodes of the changed node, which indicates the location of the changed node in an AST. We use these four attributes: *(action name, node type, label, NCS)* to represent a changed AST node.

Let's assume, there is a bug in a piece of code presented at the left side of the arrow sign in Figure 2(a). The bug resides in line number five where a developer uses `literal` '3' instead of `literal` '2'. The buggy code is fixed in the bug-fix revision, which is presented at the right side of the arrow sign in Figure 2(a). If those two revisions are given to `GumTree`, it will generate differences between the provided revisions,




(a)

(Update, NumberLiteral, 3, Infix_expression→Infix_expression→If_statement→Block→For_statement→Block→Mehod_declaration→Type_declaration→Compilation_unit)

(b)

(Update, Literal, rightOperand, 3 to 2, CtBinaryOperatorImpl→ CtBinaryOperatorImpl→CtIfImpl →CtBlockImpl→CtForImpl→CtBlockImpl→CtMethodImpl→ CtClassImpl→CtModelImpl$CtRootPackage)

(c)

**Figure 2: (a) Changing a `literal` in an `if` statement to fix a bug and the presentations of the bug-fixing change using (b) `GumTree` and (c) `GumtreeSpoon`**

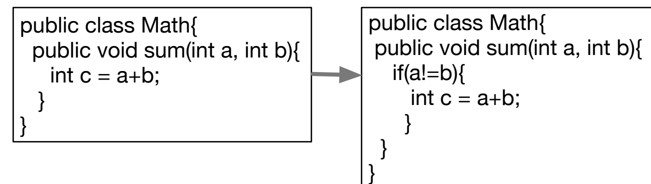
which can be presented using a tuple of four attributes as shown in Figure 2(b).

From Figure 2(b), it is easily understood that a `NumberLiteral` is *updated* to fix the bug. From the NCS (the last attribute) of the updated node, we see the `NumberLiteral` is a part of two `infix_expressions` (i.e., `==` and `%`), which reside in an `if` statement. Again, the `if` statement resides in a block under a `for` statement. The `for` statement is a part of a block inside a *method*. The method resides inside a *type declaration* (i.e., a class) and *compilation unit* is always the root of an NCS. Here, it is noticeable that an NCS represents a *sequence*, where the root and all internal nodes have only one child except the leaf node, which has no child.

### 2.3.2 Interpreting of GumtreeSpoon's output

While the output of `GumtreeSpoon` is almost similar to that of `GumTree`, there are some fundamental differences exist between their outputs. First, `GumtreeSpoon` provides a changed node's role in its immediate parent or node (i.e., *role in parent*), which helps in understanding code changes' patterns. For example, `GumtreeSpoon` indicates that the changed `literal`'s role in its parent is *rightOperand*. How the attribute *role in parent* helps in determining bug-fixing edit patterns is elaborately described in Algorithm 1 presented latter in this paper.

Second, `GumtreeSpoon` provides *modified source code* as opposed to the *label* provided by `GumTree`. We find modified source code is more helpful to understand bug-fixing edit patterns (see Section 3.2) instead of a label. Thus, we use a tuple of five attributes such as (*action name, node type, role in parent, modified source code, nested code structure*) to represent a code change using `GumtreeSpoon`'s output as shown in Figure 2(c). It is noticeable in Figure 2(b) and 2(c) that naming conventions of the nodes are different between `GumTree` and `GumtreeSpoon`.




(a)

(Insert, If, Statement, if (a != b) { ; }, CtBlockImpl→CtMethodImpl→CtClassImpl→CtModelImpl$CtRootPackage)

(b)

**Figure 3: (a) Adding an `if` statement as a precondition to fix a bug, (b) corresponding representation of the bug-fixing change using `GumtreeSpoon`**

Finally, while `GumTree` provides fine-grained level differences, `GumtreeSpoon` generates summary/concise level outputs of code changes that help in understanding bug-fixing edit patterns conveniently. For example, the code changes shown in Figure 3(a), is represented using `GumtreeSpoon`'s output in Figure 3(b). From Figure 3(b), we see that only node `if` is inserted, thus `GumtreeSpoon` ignores other fine-grained level changes such as additions of `conditional operator` and variables (e.g., `a` and `b`). In contrast to that, `GumTree's` outputs indicate that five nodes are inserted: *insert block*, *insert ifStatement*, *insert infixExpression* (i.e., ==), and *insert simpleNames* (i.e., variables `a` and `b`). While these detailed, in-depth, and verbose outputs provided by `GumTree` are suitable to analyze nesting patterns in deeper levels to answer RQ2, the concised outputs of `GumTreeSpoon` are required for analyzing bug-fixing edit patterns to answer RQ1.

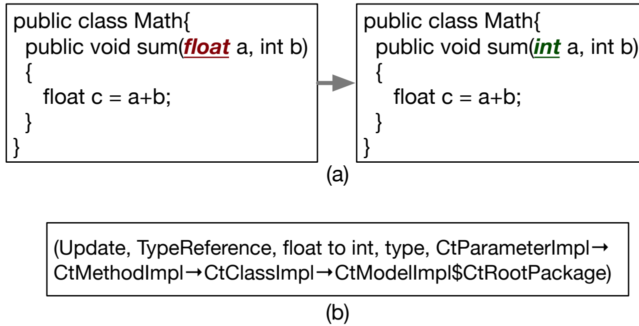



**Figure 4: (a) Updating parameter type (`float` to `int`) of a method to fix a bug and (b) corresponding representation of the bug-fixing change using `GumtreeSpoon`**

# 3. CAPTURING THE EDIT PATTERNS

Once we have the `GumtreeSpoon`'s outputs for the bug-fixing changes, we aim to identify the bug-fixing edit patterns defined by Pan et al. [57]. Pan et al. have defined a set of 27 bug-fixing edit patterns divided in nine categories: If-related (IF), Method Calls (MC), Sequence (SQ), Loop (LP), Assignment (AS), Switch (SW), Try (TY), Method Declaration (MD) and Class Field (CF).

Their study has identified the highest number of bug-fixing edit patterns in a single study. Moreover, according to the number of citations, this is one of the most important papers on bug-fix edit patterns, thus it becomes a benchmark for the studies related to bug-fixing edit patterns' detection. In the rest of this paper, we use the term *PanPattern* to refer to a pattern identified by Pan et al. [57]. We also verify whether `GumtreeSpoon` is able to identify any new bug-fixing edit patterns as opposed to the PanPatterns in our dataset.

## 3.1 Processing GumtreeSpoon's Output

While a portion of `GumtreeSpoon`'s outputs are readily interpretable, in most cases, we need to further analyze each individual NCS to detect bug-fixing edit patterns. Thus, we further process/manipulate the `GumtreeSpoon`'s output using Algorithm 1, to make those more obvious for our analysis.

Based on a preliminary investigation, we find that a code change belongs to or impacts the node that is an immediate previous node of the first occurrence of a `block` node in an NCS. For example, from the bug-fixing change presented in Figure 2(a), it is not obvious that the change occurs in an `if` statement until we see the immediate previous node of the first `block` node (which is indeed an `if` node) in the NCS given in Figure 2(c) generated by `GumtreeSpoon`.

When an NCS contains at least one `block` node, we determine the pattern of a bug-fixing change using the procedure described in Algorithm 1, Lines 2–8. An NCS starts with a `block` node if an insertion or deletion or update is performed on a node, which is not contained in or associated or linked with any other node within its block. As shown in Figure 3(b), a node `If` is inserted and the NCS starts with a `block` node. The pattern for this type of bug-fix changes is determined using the action (i.e., ins/del/upd) performed on a node to change code, and the name of the changed node as shown in Algorithm 1, Lines 3–4.

Another category of bug-fix changes contains those type of patterns where the implementation of a node is updated by performing an action on any other nodes, which are contained in or associated or linked with the implementing node within its block. As shown in Figure 2(a), a `literal` node, which is contained in an implementing `if` node, is updated where both the nodes (i.e., `if` and `literal`) reside in the same block. In this case, the bug-fixing edit pattern is determined using action, changed node name, and the immediate previous node's name of the first occurrence of a `block` node in an NCS (see Algorithm 1, Lines 6–7).

The third category of bug-fix edit patterns does not have any `block` node in the NCSes for changes in the definitions of class or interface members such as addition/removal of class fields or methods or changes in the types of parameters of methods. As shown in Figure 4(a), a developer updates type of a parameter from `float` to `int` to fix a bug. Figure 4(b) represents the change using the output of `GumtreeSpoon` where the NCS does not have any `block` node. For this case, we identify a bug-fixing edit pattern by incorporating a changed node's *role in parent* attribute, and consider the first node in the NCS as the location of the change.

If a class member (e.g., method, variable) or a parameter of a method is changed, we use action, node name, and the first node in the NCS to determine the pattern of the bug-fix change (see Algorithm 1, Lines 10–14). If the type a class variable or method's parameter is changed, then we determine the location of the change (e.g., type of a class variable or method's parameter) (see Algorithm 1, Line 16), and use that along with action and node name to determine the bug-fix pattern (see Algorithm 1, Line 17).

For the bug-fixing changes presented in Figure 2(a), Figure 3(a), and Figure 4(a), Algorithm 1 will output the patterns *update literal of CtIfImpl*, *insert `if`*, and *update type of a parameter of a method*, respectively. The set of patterns that we identify using Algorithm 1 are termed as *GSPatterns* in the rest of this paper.

## 3.2 Mapping GSPatterns to PanPatterns

In most cases, a GSPattern can be mapped directly to its corresponding PanPattern. For example, the GSPattern *update literal of CtIfImp* indicates its corresponding PanPattern *change of `if` condition expression* (IF-CC) [57].

However, we have to leverage the attribute *"modified source code"* to accurately identify PanPatterns from their corresponding GSPatterns in two cases that include: (i) addition of a precondition (i.e., `if` node) check with/without jump statement (e.g., `return`, and `break`) and (ii) changes in a method call.

In the first case, we leverage modified source code to identify whether an inserted 'If' statement acts as a precondition or not. An inserted `if` statement acts as a precondition, if it wraps up existing code, otherwise, that will be considered as a new insertion of an `if` node.

**Algorithm 1:** Detection of GSPatterns

```
   Input: T : a tuple of five attributes generated by
          GumtreeSpoon for a code change
1  String pattern;
2  if T.NCS.contains("Block") then
3  |  if T.NCS.startsWith("Block") then
4  |  |  pattern←T.action+" "+ T.nodeName;
5  |  else
6  |  |  String
   |  |   IPN←getPreviousNodeOfFirstBlock(T.NCS);
7  |  |  pattern←T.action +" " + T.nodeName + " of
   |  |   "+ IPN;
8  |  end
9  else
10 |  String FNN←getFirstNodeInNCS(T.NCS);
11 |  if T.roleInParent.equals("typeMember") then
12 |  |  pattern←T.action +" "+ T.nodeName+" in "
   |  |   + FNN;
13 |  else if T.roleInParent.equals("parameter") then
14 |  |  pattern←T.action +" "+ T.nodeName+" in "
   |  |   + FNN;
15 |  else if T.roleInParent.equals("type") then
16 |  |  String CFL←getChangeLocation(T.NCS);
17 |  |  pattern←T.action +" "+ T.nodeName +" in "
   |  |   + CFL;
18 end
19 return pattern;
```

For any inserted `if` node, if we find modified source code contains any lone semicolon (;) in a line, then the inserted `if` statement/node is considered as a precondition. For example, the modified source code, presented in Figure 3(b), contains a lone semicolon in the bug-fixing change presented in Figure 3(a). The number of such lone semicolons indicates the number of lines wrapped up by a precondition. In addition to semicolons, we also check whether the modified source code contains any jump statement such as `return`, `continue`, or `break` to identify if any precondition is added with a jump that corresponds to another PanPattern *addition of a precondition check with a jump* (IF-APCJ).

We use the same logic to identify if a piece of code is wrapped up by statements, such as `try-catch`, `loop`, or `switch-case`. We hypothesize that an inserted `if` is added as post-condition, if that is not a precondition.

In the second case, we parse modified source code to extract the method call statements in a buggy revision and its non-buggy revision. Then, for each method call, we extract the method name, arguments, and class name of a method call if available. Then, we compare that extracted information between the buggy and non-buggy method call statements to identify the location where a change occurs to map the change to its corresponding PanPattern. Multiple changes may occur in a method call (e.g., *the return type* can be changed and *an argument can be inserted*) to fix a buggy method call. In such cases, we record all types of changes and use those to identify bug-fixing edit patterns.

## 3.3 Dominant Bug-fixing Edit Patterns

In this section, we identify and describe the dominant bug-fixing *edit* patterns, which includes the most frequent *edits* found to have been applied for fixing bugs.

### 3.3.1 Detected PanPatterns

By processing `GumtreeSpoon`'s outputs we are able to detect 21 types of PanPatterns distributed in seven categories presented in Table 2. The abbreviations/initials of the categories and patterns' names are given in the same table. The MD category contains the highest number of bug-fixing changes (33.00%), followed by the IF (20.78%), MC (20.00%), and CF (16.00%) categories. Noticeable, the first four categories consist of almost 90% of bug-fixing changes. Category SW experiences the lowest number of bug-fixing changes (0.21%) preceded by LP and TY categories that consist of only 2.34% and 2.24% of the total number of PanPatterns, respectively.

The pattern MD-CHG experiences the highest number of bug-fixing changes (17.54%) followed by the patterns MC-DNP (10.24%) and IF-APTC (8.42%). Interestingly, those three patterns are from three distinct categories. MD-ADD, CF-CHG, and MD-RMV are the next three patterns that experience the highest number of bug-fixing changes (range from 7.05% to 8.17%) after those formerly mentioned three patterns.

The patterns MC-DM and IF-RMV experience almost equal amount of bug-fixing changes (≈06.70%). Surprisingly, the patterns IF-APCJ, IF-RBR and IF-ABR from IF-related category together contribute only 1.04% of the total PanPatterns. Except for the patterns SW-ARSB and TY-ARCB (that contribute only 0.21% and 0.15% of the total number of PanPatterns, respectively), the proportions of the remaining PanPatterns range from 1.32% to 5.77%.

### 3.3.2 New bug-fixing edit patterns

Using `GumtreeSpoon` we identify 17 types of new bug-fixing edit patterns in 11 categories presented in Table 3. Here, we indicate those bug-fixing edit patterns as new, which are not defined in PanPatterns. Although some of those 17 bug-fixing edit patterns are already identified in different studies [46, 47, 63], we discover four completely new/novel bug-fixing edit patterns, which were never reported before in literature. These four novel bug-fixing edit patterns are CT-AD, CT-Param, CA-AD, and EN-AD, as marked in Table 3 with bold font and asterisks.

In the following, we briefly describe the new patterns, some of which are relatively complex, while the rest others can be interpreted from their names.

**Addition or deletion of node $N_1$ ($N_1$-$AD$).** This type of pattern consists of addition or deletion of a node $N_1$ where $N_1 \in$ {`constructor`, `throw`, `loop`, `enum`, `return`, `local variable`, `assignment`}. For example, in Figure 5, we see a `constructor` is inserted in a class to fix a bug. Again, in Figure 6, a `throw` statement is deleted to fix another bug.

For each of the seven nodes, we define seven patterns such as (i) addition or deletion of `constructor` (CT-AD), (ii) addition or deletion of `throw` (TW-AD), (iii) addition or

**Table 2: Distributions of identified *PanPatterns***

| Category (Cat.) | Pattern Name | # | % | Cat. Total | Cat. % |
|---|---|---|---|---|---|
| Method Declaration (MD) | Change of method declaration (MD-CHG) | 4,116 | 17.54% | 7,687 | 33.00% |
| | Addition of a method declaration (MD-ADD) | 1,916 | 8.17% | | |
| | Removal of a method declaration (MD-RMV) | 1,655 | 7.05% | | |
| If-related (IF) | Addition of post-condition check (IF-APTC) | 1,975 | 8.42% | 4,877 | 20.78% |
| | Removal of an `if` predicate (IF-RMV) | 1,588 | 6.77% | | |
| | Change of `if` condition expression (IF-CC) | 761 | 3.24% | | |
| | Addition of precondition check (IF-APC) | 309 | 1.32% | | |
| | Addition of precondition check with jump (IF-APCJ) | 37 | 0.16% | | |
| | Removal of an `else` branch (IF-RBR) | 71 | 0.30% | | |
| | Addition of an `else` branch (IF-ABR) | 136 | 0.58% | | |
| Method Call (MC) | Method call with different number of parameters or different types of parameters (MC-DNP) | 2,402 | 10.24% | 4,639 | 20.00% |
| | Change of method call to a class instance (MC-DM) | 1,582 | 6.74% | | |
| | Method call with different actual parameter values (MC-DAP) | 655 | 2.79% | | |
| Class Field (CF) | Addition of a class field (CF-ADD) | 1,355 | 5.77% | 3,735 | 16.00% |
| | Change of class field declaration (CF-CHG) | 1,719 | 7.33% | | |
| | Removal of a class field (CF-RMV) | 661 | 2.82% | | |
| Assignment (AS) | Change of `assignment` block expression (AS-CE) | 1,401 | 0.97% | 1,401 | 5.97% |
| Loop (LP) | Change of `loop` predicate (LP-CC) | 549 | 2.34% | 549 | 2.34% |
| Try (TY) | Addition/removal of `try` statement (TY-ARTC) | 491 | 2.09% | 526 | 2.24% |
| | Addition/removal of a `catch` block (TY-ARCB) | 35 | 0.15% | | |
| Switch (SW) | Addition/removal of `switch` block branch (SW-ARSB) | 50 | 0.21% | 50 | 0.21% |
| | | | **Overall total** | **23,464** | **100%** |

**Table 3: Distributions of *newly* identified edit patterns**

| Category | Pattern Name | # | % | Cat. Total | Cat. % |
|---|---|---|---|---|---|
| Local Variable (LV) | Update implementation of `local variable` (LV-IMPL) | 4,043 | 15.41% | 7,709 | 29.38% |
| | Addition or deletion of `local variable` (LV-AD) | 3,666 | 13.97% | | |
| Method Call (MC) | Class/target change of method call (MC-TC) | 2,881 | 10.98% | 7,531 | 28.70% |
| | Addition of new method call (MC-A) | 2,754 | 10.50% | | |
| | Deletion of new method call (MC-D) | 1,896 | 7.23% | | |
| Return (RT) | Update implementation of `return` statement (RT-IMPL) | 3,361 | 12.81% | 4,200 | 16.01% |
| | Addition or deletion of `return` statement (RT-AD) | 839 | 3.20% | | |
| Assignment (AS) | Addition or deletion of `assignment block` statement (AS-AD) | 3,390 | 12.92% | 3,390 | 12.92% |
| Constructor (CT) | *Addition or deletion of `constructor` **(CT-AD)*** | 578 | 2.20% | 1,013 | 3.86% |
| | *Parameter update in `constructor` **(CT-Param)*** | 435 | 1.66% | | |
| Throw (TW) | Update of implementation of `throw` statement (TW-IMPL) | 651 | 2.42% | 861 | 3.28% |
| | Addition or deletion of `throw` statement (TW-AD) | 210 | 0.80% | | |
| Class or Interface (CI) | Addition or deletion of class or interface (CI-AD) | 480 | 1.83% | 480 | 1.83% |
| Wrap/Unwrap Code (WU-Code) | Wrap/unwrap code with/from high-level Node (WU-Code) | 410 | 1.54% | 410 | 1.54% |
| Loop (LP) | Addition and/or deletion of `loop` statement (LP-AD) | 405 | 1.54% | 405 | 1.54% |
| Catch (CA) | *Addition or deletion of `catch` variable **(CA-AD)*** | 130 | 0.50% | 130 | 0.50% |
| Enum (EN) | *Addition or deletion of `enum` statement **(EN-AD)*** | 112 | 0.43% | 112 | 0.43% |
| | | | **Overall total** | **26,241** | **100%** |

***Entirely new/novel bug-fixing edit patterns, which were never reported before.**

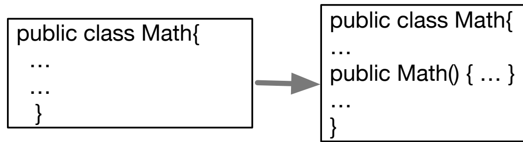



**Figure 5: Insertion of a `constructor` to fix a bug**

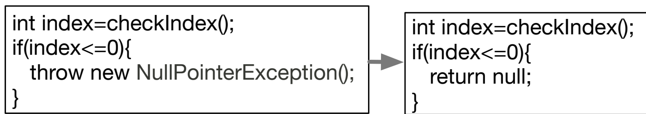



**Figure 6: Deletion of a `throw` statement to fix a bug**

deletion of `loop` (LP-AD), (iv) addition or deletion of `enum` (EN-AD), (v) addition or deletion of `return` (RT-AD), (vi) addition or deletion of `local variable` (LV-AD) and (vii) addition or deletion of `assignment` (AS-AD).

**Update implementation of node $N_2$ ($N_2$-$IMPL$).** In this type of pattern, the implementation of a node $N_2$ is updated by performing actions on other nodes associated with the implementing node. For example, in Figure 7, we see the implementation of a node `throw` is changed by updating an associated node `NullPointerException()` to `IndexOutOfBoundsException()` to fix a bug. Here $N_2 \in$ {`throw`, `return`, `local variable`}. Again, for each of the three nodes, we define three patterns, such as (i) update implementation of `throw` (TW-IMPL), (ii) update implementation of `return` (RT-IMPL), and (iii) update implementation of a `local variable` (LV-IMPL).

**Class/target change of method call (MC-TC)**. This pattern contains those types of bug-fixing changes where the class or target of a method call is changed to fix a bug. As shown in Figure 8, the owner-class of a method `getSum` is changed to fix a bug.

**Parameter update in Constructor (CT-Param)**. Similar to pattern MD-CHG, parameters of a `constructor` are often be altered to fix bugs and such changes belong to this pattern.

**Wrap/unwrap code with/from high-level Node (WU-Code)**. This pattern of code changes consists of wrapping or unwrapping existing code with/from high-level nodes. The set of high-level nodes $h$ includes {`if`, `for`, `foreach`, `while`, `do-while`, `synchronized`, `try-catch`} that can contain other types of nodes. As shown in Figure 9 a piece of existing code is wrapped up inside a `for` loop to fix a bug.

**Method call addition (MC-A) or deletion (MC-D)**. Although PanPatterns consider an addition or a deletion of a method in a sequence pattern, it is not always the case that such an addition or a deletion will be always a part of a sequence. Hence, we consider these as new patterns, which include a method's addition or deletion in a sequence.

#### 3.3.3 Comparative frequencies of the new patterns

As shown in Table 3, the category LV consists of the highest number of bug-fixing changes (29.38%) followed by the categories MC (28.70%), RT (16.01%), and AS (12.92%). Again, these four categories consist of almost 90% of newly identified bug-fixing changes.

The pattern LV-IMPL experiences the highest number of bug-fixing changes (15.41%) followed by the pattern LV-AD (13.97%). The patterns AS-AD and RT-IMPL experience almost an equal amount of bug-fixing changes (≈13%). The next three patterns MC-TC, MC-A, and MC-D are from MC categories experience 10.98%, 10.50%, and 7.23% bug-fixing changes, respectively. Those seven patterns together contribute almost 84% of bug-fixing changes. The patterns EN-AD, CA-AD, and TW-AD represent the three lowest bug-fixing changes (below 1.00%). The amounts of the rest of the patterns range from 1.50% to 3.20%.

## 4. DOMINANT NESTING PATTERNS

In Figure 10, we depict the steps required to detect nesting patterns by capturing the NCSes that frequently host the bug-fixing edits. The steps are briefly described in the following subsections.

### 4.1 Mining Patterns of Nested Structures

In Section 2.3, we see that an NCS or parents' tree structure hosting a bug-fixing edit can be presented as a *sequence* of parent nodes. Thus, to identify nesting patterns (i.e., dominant NCSes), we use a sequential pattern mining technique. Sequential pattern mining identifies a set of subsequences or patterns that occur in some percentage or, with minimum support of the input sequences. Any patterns that are found to have support values above or equal to the value of minimum support are said to be dominant patterns. Here, using a sequential pattern mining algorithm, we identify the nesting patterns that are dominant.

However, since a frequent long sequence contains a combinatorial number of frequent subsequences, such mining will generate an exhaustive set of patterns, which will be highly expensive in terms of time and space. To reduce the number of smaller sub-patterns that are found by the sequential pattern mining algorithm, we require that a mining algorithm produces *closed* or *maximal patterns* [21, 68], where sub-patterns that are contained within longer patterns are ignored.

Closed sequential pattern mining algorithms ignore all such sub-patterns that exist within other identified longer patterns and occur at the same support level [68]. On the other hand, maximal pattern mining algorithms ignore sub-patterns no matter what are their support levels [21]. As we aim to identify nesting patterns of bug-fixing changes, we find the maximal pattern mining is preferable in our case. In addition, we will apply a *gap constraint* to allow the maximum amount of gap between two nodes in the dataset of sequences to be mined.

While there are few commonly used sequential pattern mining algorithms available [20], we use the recently proposed `MG-FSM` algorithm [49] that meets our requirement to specify constraints such as pattern type (e.g., closed or maximal) and gap constraint between two successive nodes. Moreover, the algorithm is capable of parallel running using map-

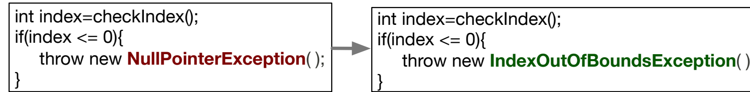



**Figure 7: Altering a `throw` statement to throw a different exception**

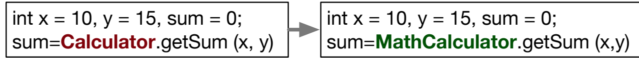



**Figure 8: Changing class/target of a method call to fix a bug**

reduce (Hadoop) functionality and it is suitable in using on a cloud infrastructure having the capacity to deliver the desired scalability for larger datasets [49].

We run the tool by allowing no gap between two successive nodes to determine maximal patterns that have at least 1,000 occurrences. The tool delivers a total of 534 sequences that are dominant. We exclude those patterns that do not have at least one block node to make sure containment of a node inside another node. Finally, we have 385 nesting patterns that we use for clustering as follows.

## 4.2 Clustering of Nesting Patterns

At this step, we cluster similar types of nesting patterns in groups. Such clustering provides a convenient way to examine the identified nesting patterns where developers commonly perform bug-fixing changes.

### *4.2.1 Selection of the clustering algorithm*

To cluster nesting patterns, we use the k-medoids [36] algorithm that is a variant of the k-means [35] algorithm. While both the k-means and k-medoids algorithms break a dataset up into groups, the latter algorithm uses existing points in the dataset as cluster centroids. Moreover, k-medoids is known for more robustness against noises and outliers compared to k-means [36]. In addition, in our dataset k-means cannot be used directly because numerical operations, such as addition and division, cannot be performed on two patterns, which consist of strings [49].

### *4.2.2 Determining the optimal number of clusters*

To determine the optimal number of clusters (i.e., k in k-medoids), we choose to use the *gap statistic* [66] method over other available options such as *elbow* and *silhouette* methods. The reason why we choose this method as it can be applied to any clustering method (i.e., k-medoids, k-means clustering, and hierarchical clustering). Using the gap statistic, we find the number of optimal clusters is 10 for our data.

The gap statistic compares the total intra-cluster variation for different values of k with their expected values under null reference distribution of the data (i.e. a distribution with no obvious clustering). The reference dataset is generated using *Monte Carlo* [48] simulations of the sampling process. The details of the gap statistic method can be found elsewhere [66].

### *4.2.3 Defining a distance function for k-medoids*

We use the *Longest Common Subsequence* (LCS) based string metric to measure the distance between a pair of mined nesting patterns. We define the distance function for any two mined nesting patterns $S_1$ and $S_2$ as follows.

$$D_{LCS}(S_1, S_2) = 1 - \frac{|\ LCS(S_1, S_2)\ |}{max(|\ S_1\ |, |\ S_2\ |)}$$

Here, $S_1$ and $S_2$ are two finite sequences of distinct nodes, $|\ LCS(S_1, S_2)\ |$ is the length of the longest common subsequence(s) of $S_1$ and $S_2$, and $max(|\ S_1\ |, |\ S_2\ |)$ is the length of the longest sequence of $S_1$ and $S_2$.

A value of 0.0 for the distance function $D_{LCS}(S_1, S_2)$ indicates that two patterns are identical, while a value of 1.0 indicates that the patterns are completely different.

We measure the LCS metric by using the Python package `python-string-similarity` [10] that implements the said metric. Then, to cluster nesting patterns, we run the open-source implementation of the k-medoids algorithm provided in Python clustering package `Pycluster 1.49` [9].

At this point, we have 10 clusters of nesting patterns. As the mechanism to generate cluster is based on the names of the AST nodes (i.e., text-based clustering), the clusters are required to be interpreted/characterized by human experts to gain meaningful insights of the structures of the nesting patterns in the clusters.

## 4.3 Characterization of the Clusters

The first two authors are presented with a listing of all the patterns in each cluster and asked to characterize those patterns in terms of their nodes' hierarchies. By observing nodes' hierarchies of the patterns, they create two sets of nodes: (i) a set of low-level nodes $l$, where $l \epsilon$ {**`return`**, **`expression`**, **`throw`**, **`variable declaration`**, **`assignment`**} and (ii) another set of high-level nodes $h$ defined in Section 3.3.2.

Each author aims to identify if a low-level node's block from $l$ is contained in a high-level node block from $h$. Such identification is represented as a pattern/cluster $l$ block→$h$ block. For example, if a block of **`return`** is located inside an **`if`** block, then the authors label that hierarchy as **`return`** block→**`if`** block. If a higher-level node block $h_1$ is contained in another high-level node block $h_2$, then that pattern is categorized as 'Compound' and represented as $h_1$ block→$h_2$ block. In a similar fashion, deeper-levels' containments/hierarchies can also be presented (see the third column of the last row in Table 4).

Cohen's kappa coefficient $\kappa$ [15] is used to measure agreement between two authors in characterizing patterns. $\kappa$ value 0.79 indicates high-level agreement on the characterization of the patterns of the clusters. For each disagreement,

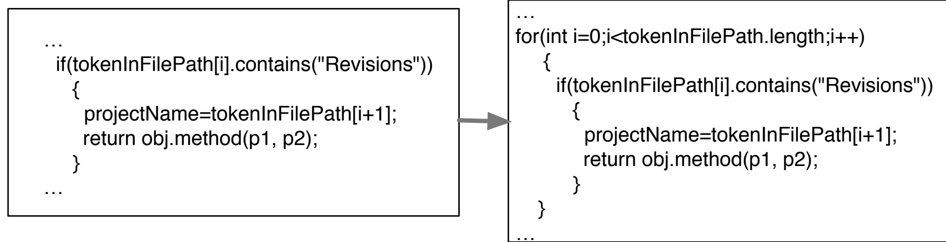



**Figure 9: Wrapping up existing code using a `for` loop to fix a bug**

**Table 4: Dominant *Nesting* Patterns that frequently host bug-fixing edits**

| **Category (Cat.)** | **Cluster/ Pattern ID** | **Mined Nesting Patterns** | **#** | **%** | **Cat. Total** | **Cat. %** |
|---|---|---|---|---|---|---|
| IF-related (IF) | 01 | `if` block→`if` block | 27,306 | 10.93% | 101,691 | 40.72% |
| | 02 | Method invocation block→`if` block | 19,430 | 7.78% | | |
| | 03 | `expression` block→`if` block | 16,838 | 6.74% | | |
| | 04 | `assignment` block→`if` block | 10,250 | 4.10% | | |
| | 05 | `variable declaration` block→`if` block | 9,996 | 4.00% | | |
| | 06 | `throw` block→`if` block | 9,274 | 3.71% | | |
| | 07 | `return` block→`if` block | 4,923 | 1.97% | | |
| | 08 | Method invocation block as `expression` →`if` block | 3,771 | 1.51% | | |
| | 09 | Nested `If` (with/without `else`) | 2,634 | 1.05% | | |
| Try-Catch (TY-CA) | 10 | block→`try` block | 12,321 | 4.93% | 24,709 | 9.89% |
| | 11 | `variable declaration` block→`catch` block | 5,858 | 2.34% | | |
| | 12 | Method invocation block→`try` block | 3,197 | 1.28% | | |
| | 13 | `throw` block→`try-catch` block | 1,248 | 0.49% | | |
| | 14 | `expression` block →`try` block | 1,048 | 0.41% | | |
| | 15 | `try` block→`try` block | 1,037 | 0.41% | | |
| Loop (LP) | 16 | `variable declaration` block→`loop` block | 10,202 | 4.08% | 20,029 | 8.02% |
| | 17 | Method invocation block→`loop` block | 6,001 | 2.40% | | |
| | 18 | `expression` block→`loop` block | 2,316 | 0.92% | | |
| | 19 | `assignment` block→`loop` block | 1,510 | 0.60% | | |
| Chained Method Invocations (CMI) | 20 | Chained method invocations | 14,828 | 5.93% | 14,828 | 5.93% |
| Synchronize (SYN) | 21 | `if` block→`synchronized` block | 4,888 | 1.95% | 8,986 | 3.60% |
| | 22 | `loop` block→`synchronized` block | 4,098 | 1.64% | | |
| Compound (COM) | 23 | `if` block→`loop` block | 27,583 | 11.04% | 79,484 | 31.82% |
| | 24 | `if` block→`try` block | 13,328 | 5.33% | | |
| | 25 | `loop` block→`if` block | 6,457 | 2.58% | | |
| | 26 | `loop` block→`try` block | 4,818 | 1.92% | | |
| | 27 | `try` block→`if` block | 3,725 | 1.49% | | |
| | 28 | `expression` block→`loop` block→`if` block | 3,687 | 1.47% | | |
| | 29 | `loop` block→`loop` block | 3,386 | 1.35% | | |
| | 30 | `try` block→`loop` block | 2,205 | 0.88% | | |
| | 31 | `if` block→`loop` block→`if` block | 2,141 | 0.85% | | |
| | 32 | `switch` block→`if` block | 1,413 | 0.56% | | |
| | 33 | `if` block→`loop` block→`try` block | 1,189 | 0.47% | | |
| | 34 | `if` block→`switch` block | 1,161 | 0.46% | | |
| | 35 | `switch` block→`loop` block | 1,145 | 0.45% | | |
| | 36 | `variable declaration` block→`if` block→`loop` block | 1,118 | 0.44% | | |
| | 37 | `expression` block→`loop` block→`try` block | 1,026 | 0.41% | | |
| | | | **Overall Total** | | **249,727** | 100% |

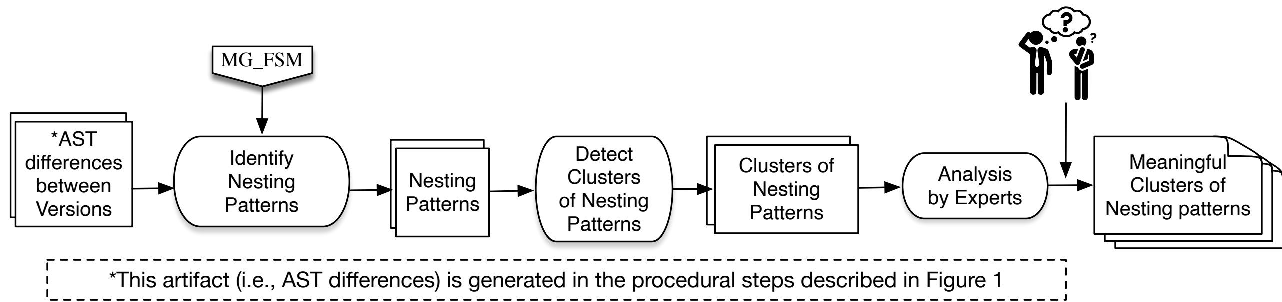



**Figure 10: Steps to identify nesting patterns that frequently host bug-fixing edits**

authors discuss between them, and if necessary, they verify raw data to come to an agreement. Such discussions result in an unconventional pattern that has chained method invocations (e.g., m1().m2().m3()) in a single block (see the fourth category in Table 4).

Finally, total of 37 meaningful clusters is identified in six categories: (i) IF-related (IF), (ii) Try-Catch (TY-CA) (iii) Loop (LP) (iv) Chained Method Invocation (CMI) (v) Synchronize (SYN) and (vi) Compound (COM) as presented in Table 4. As per the definition of the category COM, the category SYN falls in the COM category, although the authors decide to create a separate category for it. As all the patterns are found to have ended with a common suffix, `Method_declaration`→`Type_declaration`→`Compilation_unit`, we truncate that for cleaner presentation.

## 4.4 Mining Results

There are nine types of nesting patterns or clusters belong to the IF category that represents the largest amount (40.72%) of the total number of patterns followed by the COM category. The COM category consists of 15 types of patterns contribute to 31.82% of the total number of patterns. The categories TY-CA and LP contribute 9.89% and 8.02% of total patterns, respectively, followed by the CMI category that consists of 5.93% of total patterns. The number of patterns belongs to the SYN category is the lowest (3.60%).

By inspecting individual clusters, we find some interesting patterns that can not be identified without considering hierarchies of NCSes. The 23rd cluster (i.e., `if` block→`loop` block) is the most bug-prone pattern as it experiences the highest number (27,583) bug-fix changes followed by the first cluster `if` block→`if` block, which is slightly lower than the former cluster. Noticeable, the number of bug-fix changes in a pattern $l$ block→`if` block is always higher than a pattern $l$ block→$h'$ block, where $h' = h-$`if`. For example, the number of occurrences of the pattern `expression` block→`if` block is higher than the number of occurrences of the pattern `expression` block→`loop` block. Recalling that $l$ represents the set of low-level nodes.

It is very interesting that `throw` blocks inside `if` blocks are more bug-prone than `throw` blocks inside `try-catch` blocks. Although in Table 2 we see the number of changes in the category Try (TY) is very low, the opposite result is observed in Table 4, where category related to Try (TY-CA) experiences the second highest bug-fix changes among the categories of simple high-level nodes. It means that pieces of code inside `try-catch` frequently experience bug-fix changes. A similar observation is also applicable to the SYN category.

Surprisingly, only five clusters among 37 clusters contain three-level containment (see clusters 28, 31, 33, 36, and 37), which consist of only 3.64% of all patterns. The pattern `expression` block→`loop` block→`if` block consists of almost 50% of all those three-levels patterns. No pattern is found that contains more than three-levels of containment.

# 5. THREATS TO VALIDITY

This section discusses the possible limitations of this work, the threats to the validity of the findings, and our efforts to minimize those threats and shortcomings.

## 5.1 Construct Validity

For the first three projects, we use the bug-fixing commits that are identified by Ray et al. [59]. To collect those buggy commits, they used a technique similar to the approach of Mockus and Votta [50]. A similar approach is also used for detecting bug-fixing commits of the last two projects. There is a possibility that some portions of the bug-fixing commits may be general commits (e.g., new feature, and improvement), thus, our data may contain some false positives. Having said that, Ray et al. found 96% accuracy of their approach to collect those bug-fixing commits that minimized the threat substantially.

To detect bug-fixing edit patterns, we have considered only nodes found before the occurrence of the first block node (if available) in an NCS. Someone may be skeptical of the capabilities of detecting bug-fixing edit patterns using such an approach. However, our approach is found successful in detecting not only existing bug-fixing edit patterns but also new bug-fixing edit patterns from code bases.

To detect nesting patterns, we have identified maximal patterns instead of closed patterns of nested code structures. Closed sequential pattern mining algorithms remove all patterns that exist within other identified patterns and occur at the same support level, while maximal pattern mining removes sub-patterns regardless of the support level. For our problem, the maximal pattern mining is preferable, as we have aimed to identify deeper nested code structures instead of sub-structures (i.e., sub-patterns).

To detect maximal patterns, we have allowed no gap between two nodes and only considered those as patterns, which have at least 1,000 occurrences. To detect exact nested code structures, it is obvious that setting “no gap between two nodes” is the best choice. Although the setting of 1,000 as the threshold can be criticized, we have found the setting was capable of retaining over 70% bug-fixing transactions, while minimized human efforts in detecting meaningful clusters.

In detecting patterns, we have excluded any *mov* actions on nodes found in ASTs for bug-fixing changes. Typically, an insertion or a deletion of a node caused moving of other nodes in an AST where moved nodes remain unchanged. Thus, such unchanged nodes (i.e., *mov* actions) can be ignored, although it is still a threat in detecting patterns related to *mov* action.

## 5.2 Internal Validity

The correctness of our analysis depends on both `GumTree` and `GumtreeSpoon` tools, which are used to answer *RQ2* and *RQ1*, respectively. The former tool outperforms the state-of-the-art tool `ChangeDistiller` by maximizing the number of AST node mappings, minimizing the edit script size, and detecting better move actions [17]. Moreover, `ChangeDistiller` works at the statement level, preventing the detection of certain fine-grain patterns.

Similar to `GumtreeSpoon`, there is another tool `CLDiff` [26] also available. Between `CLDiff` and `GumtreeSpoon`, we selected the latter tool for addressing *RQ1*, because, from a sample test run, we revealed that `CLDiff` failed in distinguishing changes to method parameters at its concise level outputs. But, for this work on bug-fixing edit patterns, this capability is very important for capturing subtle changes in a code made for bug-fixing [57].

The library `JGit` [5] is used in our work to extract a buggy and its previous revisions. The library is applied for a similar purpose in other studies [23, 55], which has brought confidence in us to use that.

## 5.3 External Validity

Although our study includes a large number of revisions of five subject systems, all the systems are open-source and written in Java. Thus, the findings from this work may not generalize for industrial systems and source code written in languages other than Java.

## 5.4 Reliability

The methodology of this study including the procedures for data collection and analysis are documented in this paper. The subject systems being open-source, are freely accessible while the tools `GumTree`, `GumtreeSpoon`, `MG-FSM`, and library `JGit` are also available online. Moreover, the bug-fixing commits are also publicly available. Therefore, it should be possible to replicate the study.

# 6. RELATED WORK

In discussing the related work, we organize previous research relevant to ours in two categories. First, we discuss those studies that were specific to detecting bug-fix patterns (Section 6.1). Second, we discuss the work carried out to identify code change patterns in general, not necessarily focusing on capturing particularly the bug-fixing changes (Section 6.2).

## 6.1 Identifying Bug-Fixing Edit Patterns

Pan et al. [57] manually identified a set of 27 bug-fixing edit patterns (i.e., PanPatterns) by exploiting textual differences between buggy and non-buggy programs. A similar approach was also applied by Yue et al. [69] to identify 11 bug-fixing edit patterns, however, they used a clustering technique to minimize manual efforts. Both studies are subject to few limitations: (i) obviously identifying all possible bug-fixing edit patterns using manual effort is a daunting task, which arises the possibility of failure in discovering all types of bug-fixing edit patterns, and (ii) they used textual differences between buggy and non-buggy programs to identify bug-fixing edit patterns, which is reported to be limited in detecting bug-fixing edit patterns [17].

Despite few limitations, the study of Pan et al. is the most influential work (in terms of citation numbers) in the related area and till now they have identified the highest number of bug-fixing edit patterns in code bases. That is why we started our work by targeting this study to identify bug-fixing edit patterns (while at the same time we kept an eye on any new or unseen patterns). Instead of textual difference, we have used a state-of-the-art AST based code differencing tool `GumtreeSpoon` and developed a fully automated approach to detect bug-fixing edit patterns. Moreover, we have detected 37 bug-fixing edit patterns, which is the highest among all the studies [38, 47, 46, 57, 63] carried out to identify bug-fixing edit patterns to date. In addition, we have identified four new patterns in Constructor, Catch, and Enum categories (see Table 3).

Kim et al. [38] also employed manual efforts to identify a set of 10 dominant bug-fixing edit patterns (known as PAR templates). However, to collect bug-fixing patches they used the `Kenyon` framework [12] and clustered those patches using `groums` [54] to minimize human efforts. Again, Sobreira et al. [63] manually analyzed only 395 patches collected from *Defects4J* [37] project and identified 25 bug-fixing edit patterns. Although the latter study identified the second highest number bug-fixing edit patterns after PanPatterns and identified a few new patterns too, the work was conducted on a very small dataset.

Thus, additional studies are required to verify their newly identified bug-fixing edit patterns related to `return`, `throw` and wrap/unwarp-code using larger datasets and our work can be considered such a work that verifies those new patterns are dominant in bug-fixing changes. Moreover, they identified 33 instances of a pattern where `throw` blocks reside in `if` blocks. The sixth cluster in Table 4 confirms such finding of their study. In addition, in a very recent study, Tufano et al. [67] manually identified new patterns of only five instances related to `synchronized` blocks’ additions and deletions. The question is ”can we consider those as patterns with only five instances?”. Our study answers this question in the positive from the observation of the 21st and 22nd clusters in the SYN category presented in Table 4.

To overcome the problems of manual approaches, automatic techniques are developed based on AST code differencing tools to identify bug-fixing edit patterns. Martinez and Monperrus [47] identified 20 bug-fixing edit patterns using the AST based code differencing tool `ChangeDistiller` [19]. In our study, we have used the tool `GumtreeSpoon` [4] developed based on `GumTree` [17], which is more accurate than `ChangeDistiller`.

Soto et al. [64] conducted a study of bug-fixing commits in Java projects. Campos et al. [14] characterized the prevalence of the five most common bug-fixing edit patterns related to IF category. However, the studies of Soto et al. and Campos et al. limited the searching of bug-fixing edit patterns within PAR templates and patterns identified by Pan et al., respectively. In our case, we have remained open to identify any bug-fixing edit patterns exist in code bases. Osman et al. [55] applied a semi-automated approach to identify five bug-fixing edit patterns. In contrast to their approach, we have used a fully automated technique to identify 37 bug-fixing edit patterns.

Hanam et al. [24] developed the `BugAID` technique for discovering bug-fixing edit patterns in JavaScript. Sudhakrishnan [65] identified 25 bug-fixing edit patterns in Verilog (a hardware description language). Long and Rinard [44] learned a probabilistic model of correct patch from bug-fixing changes of C code. In contrast to their studies, we have studied five Java projects. While all the previously mentioned studies identified bug-fixing edit patterns, none of the studies conducted any nested code structures analysis to detect nesting patterns, which is a *novel contribution* of our study.

Few other studies [43, 56, 60] identified fixing patterns of violations of static good coding principles identified by tools such as `PMD` [7], `FindBugs` [16] and `Splint` [11]. However, in our study, we have studied real bug-fixing changes instead of coding principles' violations.

## 6.2 Identifying Code-Change Patterns

In the past, studies were conducted to understand the of characteristics and patterns of buggy code with respect to different criteria such as code quality, security vulnerabilities, stability [30, 31, 32, 34, 61, 70, 71]. There have also been many studies to identify patterns of code changes in general, which are not specific to bug-fixing edits. Here we discuss only those, which are the most relevant to our work.

Fluri et al. [18] identified code-change patterns in three Java applications using `ChangeDistiller`. Again, Martinez et al. [46] used `ChangeDistiller` to identify 18 code-change patterns. Molderez et al. [51] developed an automated system to mine code-change histories to detect unknown repetitive code-changes. Kim et al. [39, 41] discovered logical and structural changes at or above the method level. Breu and Zimmermann [13] extracted method call change patterns. Negara et al. [53] developed the tool `CodingTracker` and discovered previously unknown 10 code-change patterns. Kim et al. [40] developed the tool `LSdiff` that can group code-changes from systematic change patterns. While all these studies analyzed code-change patterns, we have studied bug-fixing changes by mining software repositories.

# 7. CONCLUSION

Fixing bugs in software systems involves two fundamental tasks: first, identification of the location of the bug (i.e., bug localization), and second, making edits to the existing code to fix the bug (i.e., refactoring). Towards better assistance and a deeper understanding of these two tasks, this paper identifies bug-fixing *nesting* patterns and *edit* patterns by conducting a combination of quantitative and qualitative analyses of 4,653 buggy revisions of five open-source software systems written in Java.

Our analyses have reported 38 bug-fixing *edit* patterns organized in 14 categories. This is the highest number of bug-fixing *edit* patterns identified in a single study. 34 of the 38 bug-fixing *edit* patterns, confirm those reported earlier in the literature. Four of the identified bug-fixing *edit* patterns are completely new (i.e., were never reported before). These four new *edit* patterns are related to the `constructor`, `try-catch`, and `enum` code constructs.

Among the 14 identified categories of bug-fixing *edit* patterns, Method Call (MC), Method Declaration (MD), Local Variable (LV), If-related (IF), Assignment (AS), and Return (RT) are the six most dominant categories that frequently host bug-fixing edits. The four least dominant categories are Try (TY), Switch (SW), Catch (CA), and Enum (EN). These findings are in accordance with the observations reported in other studies [57, 63, 47] in the past.

Using sequential pattern mining and clustering techniques, we have discovered 37 new bug-fixing *nesting* patterns, which capture the locations of the bug-fixing edits within the nested code structure surrounding them. This new set of *nesting* patterns is a *novel* contribution that adds a new dimension to our understanding of bug-fixing patterns. The identified *nesting* patterns are organized in six categories as listed in descending order of their frequencies as follows: If-related (IF), Compound (COM), Loop (LP), Try-catch (TY-CA), Synchronize (SYN), and Chain Method Invocation (CMI).

Our analysis of the *nesting* patterns reveals additional insights into the locations where big-fixing edits frequently take place. We have found that any nodes/blocks associated with `if` blocks are the most bug-prone. The *nesting* pattern "`if` block inside `loop` block" experiences the highest number bug-fixing edits, followed by the "`if` block inside another `if` block" *nesting* pattern. Moreover, for the first time, in this study, we have discovered that *nesting* patterns in the CMI category experience a significant number of bug-fixing edits. Our analysis of the nesting patterns also indicates nodes/blocks inside `try-catch` and `synchronized` constructs are bug-prone.

The findings from this work deepen our understanding of bug-fix patterns. Both the bug-fixing *edit* patterns and *nesting* patterns can also be useful in devising techniques for automated program repair. For example, existing probabilistic patch generation algorithms can incorporate patterns of bug-fix edits and their locations in nested code structures to maximize the probabilities of locating bugs and generating patches for those bugs successfully. Our future work will explore these possibilities.

## 8. ACKNOWLEDGEMENT

This work is supported in part by the SCoRe (**S**timulating **Co**mpetitive **Re**search) grant at the Department of Computer Science, University of New Orleans, USA.

## 9. REFERENCES

[1] *Apache Accumulo: a highly scalable open source data store*. https://accumulo.apache.org, verified: Nov 2020.

[2] *Commons Math: The Apache Commons Mathematics Library*. https://commons.apache.org/proper/commons-math/, verified: Nov 2020.

[3] *Facebook SDK for Android*. https://github.com/facebook/facebook-android-sdk, verified: Nov 2020.

[4] *GumtreeSpoon - Spoon version of GumTree*. https://github.com/SpoonLabs/gumtree-spoon-ast-diff, verified: Nov 2020.

[5] *JGit*. https://www.eclipse.org/jgit/, verified: Nov 2020.

[6] *Netty: an asynchronous event-driven network application framework*. https://netty.io, verified: Nov 2020.

[7] *PMD - Source Code Analyzer*. http://pmd.sourceforge.net, verified: Nov 2020.

[8] *Presto: Distributed SQL Query Engine for Big Data*. https://prestodb.io, verified: Nov 2020.

[9] *PyCluster - Clustering module for Python*. https://bioconda.github.io/recipes/pycluster/README.html, verified: Nov 2020.

[10] *Python-String-Similarity*. https://github.com/luozhouyang/python-string-similarity/blob/master/README.md, verified: Nov 2020.

[11] *SPLINT - Secure Programming LINT*. http://www.splint.org, verified: Nov 2020.

[12] J. Bevan, E. Whitehead, S. Kim, and M. Godfrey. Facilitating software evolution research with kenyon. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 177–186, 2005.

[13] S. Breu and T. Zimmermann. Mining aspects from version history. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, pages 221–230, 2006.

[14] E. Campos and M. Maia. Common bug-fix patterns: A large-scale observational study. In *Proceedings of the Empirical Software Enginerign and Measurement*, pages 404–413, 2017.

[15] A. Cantor. Sample-size calculations for cohen's kapp. *Psychological Methods*, 1(2):150–153, 1996.

[16] B. Cole, D. Hakim, D. Hovemeyer, R. Lazarus, W. Pugh, and K. Stephens. Improving your software using static analysis to find bugs. In *OOPSLA*, pages 673–674, 2006.

[17] J. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus. Fine-grained and accurate source code differencing. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, pages 313–324, 2014.

[18] B. Fluri, E. Giger, and H. Gall. Discovering patterns of change types. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, pages 15–19, 2008.

[19] B. Fluri, M. Wursch, M. Pinzger, and H. Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Transactions on Software Engineering*, 33(11):725–743, 2007.

[20] P. Fournier-Viger, J. Lin, A. Gomariz, T. Gueniche, A. Soltani, and Z. Deng. The SPMF open-source data mining library version 2. In *Proceedings of the Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 36–40, 2016.

[21] P. Fournier-Viger, C. Wu, A. Gomariz, and V. Tseng. VMSP: Efficient vertical mining of maximal sequential patterns. *Advances in Artificial Intelligence*, 8436:83–94, 2014.

[22] Tricentis GmbH. *Software Fail Watch: 5th Edition*. https://www.tricentis.com/resources/software-fail-watch-5th-edition/, verified: Nov 2020.

[23] G. Greene and B. Fischer. Cvexplorer: Identifying candidate developers by mining and exploring their open source contributions. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, pages 804–809, 2016.

[24] Q. Hanam, F. Brito, and A. Mesbah. Discovering bug patterns in javascript. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 144–156, 2016.

[25] M. Hashimoto and A. Mori. Diff/TS: A tool for fine-grained structural change analysis. In *Proceedings of the Working Conference on Reverse Engineering*, pages 279–288, 2008.

[26] K. Huang, B. Chen, X. Peng, D. Zhou, Y. Wang, Y. Liu, and W. Zhao. Cldiff: Generating concise linked code differences. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, pages 679–690, 2018.

[27] Research T. Institute. The economic impacts of inadequate infrastructure of software testing. RTI Project Report 7007.011, National Inst. of Standards and Tech., 2002.

[28] J. Islam, M. Mondal, and C. Roy. Bug replication in code clones: An empirical study. In *Proceedings of the International Conference on Software Analysis, Evolution, and Reengineering*, 2016.

[29] J. Islam, M. Mondal, C. Roy, and K. Schneider. A comparative study of software bugs in clone and non-clone code. In *Proceedings of the International*

*Conference on Software Engineering and Knowledge Engineering*, pages 436–443, 2017.

[30] M. Islam and M. Zibran. A comparative study on vulnerabilities in categories of clones and non-cloned code. In *Proceedings of the 10th IEEE International Workshop on Software Clones*, pages 8–14, 2016.

[31] M. Islam and M. Zibran. On the characteristics of buggy code clones: A code quality perspective. In *Proceedings of the 12th IEEE International Workshop on Software Clones*, pages 23 – 29, 2018.

[32] M. Islam and M. Zibran. Sentiment analysis of software bug related commit messages. In *Proceedings of the 27th International Conference on Software Engineering and Data Engineering*, pages 3–8, 2018.

[33] M. Islam and M. Zibran. How bugs are fixed: Exposing bug-fix patterns with edits and nesting levels. In *Proceedings of the 35th ACM/SIGAPP Symposium on Applied Computing*, pages 1523–1531, 2020.

[34] M. Islam, M. Zibran, and A. Nagpal. Security vulnerabilities in categories of clones and non-cloned code: An empirical study. In *Proceedings of the 11th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 20–29, 2017.

[35] A. jain. Data clustering: 50 years beyond k-means. *Pattern Recognition Letters*, 31(8), 2010.

[36] X. Jin and J. Han. *K-Medoids Clustering*. Encyclopedia of Machine Learning (Springer), 2011.

[37] R. Just, D. Jalali, and M. Ernst. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 437–440, 2014.

[38] D. Kim, J. Nam, J. Song, and S. Kim. Automatic patch generation learned from human-written patches. In *Proceedings of the International Conference of Software Engineering*, pages 802–811, 2013.

[39] M. Kim, J. Beall, and D. Notkin. Discovering and representing logical structure in code change. Technical report, University of Washington, 2007.

[40] M. Kim and D. Notkin. Discovering and representing systematic code changes. In *Proceedings of the International Conference on Software Engineering*, pages 309–319, 2009.

[41] M. Kim, D. Notkin, and D. Grossman. Automatic inference of structural changes for matching across program versions. In *Proceedings of the International Conference of Software Engineering*, 2007.

[42] S. Kim, K. Pan, and E. Whitehead. Memories of bug fixes. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 35–45, 2006.

[43] K. Liu, D. Kim, T. Bissyande, S. Yoo, and Y. Traon. Mining fix patterns for findbugs violations. *IEEE Transactions on Software Engineering*, pages 1–24, 2018.

[44] F. Long and M. Rinard. Automatic patch generation by learning correct code. In *Proceedings of the Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 298–312, 2016.

[45] J. Maletic and M. Collard. Supporting source code difference analysis. In *Proceedings of the IEEE International Conference on Software Maintenance*, page PP, 2004.

[46] M. Martinez, L. Duchien, and M. Monperrus. Automatically extracting instances of code change patterns with ast analysis. In *Proceedings of the IEEE International Conference on Software Maintenance*, pages 22 – 28, 2013.

[47] M. Martinez and M. Monperrus. Mining software repair models for reasoning on the search space of automated program fixing. *Empirical Software Engineering*, 20(1):176–205, 2015.

[48] Monte Carlo method. *Monte Carlo method*. https://en.wikipedia.org/wiki/Monte_Carlo_method, verified: Nov 2020.

[49] I. Miliaraki, K. Berberich, R. Gemulla, and S. Zoupanos. Mind the gap: Large-scale frequent sequence mining. In *Proceedings of the International Conference on Management of Data*, pages 797–808, 2013.

[50] A. Mockus and L. Votta. Identifying reasons for software changes using historic databases. In *Proceedings of the International Conference on Software Maintenance and Evolution*, pages 120–130, 2000.

[51] T. Molderez, R. Stevens, and C. Roover. Mining change histories for unknown systematic edits. In *Proceedings of the International Conference on Mining Software Repositories*, pages 248–256, 2017.

[52] M. Mondal, C. K. Roy, and K. A. Schneider. Identifying code clones having high possibilities of containing bugs. In *Proceedings of the IEEE/ACM International Conference on Program Comprehension*, pages 99–109, 2017.

[53] S. Negara, M. Codoban, D. Dig, and R. Johnson. Mining fine-grained code changes to detect unknown change patterns. In *Proceedings of the International Conference on Software Engineering*, pages 803–813, 2014.

[54] T. Nguyen, H. Nguyen, N. Pham, J. Al-Kofahi, and T. Nguyen. Graph-based mining of multiple object usage patterns. In *Proceedings of the ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 383–392, 2009.

[55] H. Osman, M. Lungu, and O. Nierstrasz. Mining frequent bug-fix code changes. In *Proceedings of the IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering*, pages 343–347, 2014.

[56] H. Oumarou, N. Anquetil, A. Etien, S. Ducasse, and K. Taiwe. Identifying the exact fixing actions of static rule violation. In *Proceedings of the IEEE*

*International Conference on Software Analysis, Evolution and Reengineering*, pages 371–379, 2015.

[57] K. Pan, S. Kim, and E. Whitehead Jr. Toward an understanding of bug fix patterns. *Empirical Software Engineering*, 14(3):286–315, 2009.

[58] F. Rahman, C. Bird, and P. Devanbu. Clones: what is that smell? In *Proceedings of the International Conference of Mining Software Repositiries*, pages 72–81, 2010.

[59] B. Ray, V. Hellendoorn, S. Godhane, Z. Tu, A. Bacchelli, and P. Devanbu. On the "naturalness" of buggy code. In *Proceedings of the International Conference of Software Engineering*, pages 428–439, 2016.

[60] R. Rolim, G. Soares, R. GheyI, T. Barik, and L. D'Antoni. Learning quick fixes from code repositories. In *arXiv preprint arXiv:1803.03806*, 2018.

[61] R. Saha, M. Asaduzzaman, M. Zibran, C. Roy, and K. Schneider. Evaluating code clone genealogies at release level: An empirical study. In *SCAM*, pages 87–96, 2010.

[62] R. Saha, Y. Lyu, H. Yoshida, and M. Prasad. Elixir: Effective object-oriented program repair. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, pages 648–659, 2017.

[63] V. Sobreira, T. Durieux, F. Madeiral, M. Monperrus, and M. Maia. Dissection of a bug dataset: Anatomy of 395 patches from Defects4J. In *Proceedings of the IEEE International Conference on Software Analysis, Evolution and Reengineering*, 2018.

[64] M. Soto, F. Thung, C. Wong, C. Goues, and D. Lo. A deeper look into bug fixes: Patterns, replacements, deletions, and additions. In *Proceedings of the International Conference of Mining Software Repositiries*, pages 512–515, 2016.

[65] S. Sudhakrishnan, J. Madhavan, and E. Whitehead Jr. Understanding bug fix patterns in verilog. In *Proceedings of the International Working Conference on Mining Software Repositories*, pages 39–42, 2015.

[66] R. Tibshirani, G. Walther, and T. Hastie. Estimating the number of clusters in a data set via the gap statistic. *Journal of the Royal Statistical Society*, 63(2):411–423, 2001.

[67] M. Tufano, J. Pantiuchina, C. Watson, G. Bavota, and D. Poshyvanyk. On learning meaningful code changes via neural machine translation. In *Proceedings of the International Conference of Software Engineering*, page (to appear), 2019.

[68] J. Wang, J. Han, and C. Li. Frequent closed sequence mining without candidate maintenance. *IEEE Trans. on Knowledge Data Engineering*, 19(8):1–15, 2007.

[69] R. Yue, N. Meng, and Q. Wang. A characterization study of repeated bug fixes. In *Proceedings of the International Conference on Software Maintenance and Evolution*, pages 422–432, 2017.

[70] M. Zibran, R. Saha, C. Roy, and K. Schneider. Evaluating the conventional wisdom in clone removal: A genealogy-based empirical study. In *Proceedings of the 28th ACM/SIGAPP Symposium on Applied Computing*, pages 1123–1130, 2013.

[71] M. Zibran, R. Saha, C. Roy, and K. Schneider. Genealogical insights into the facts and fictions of clone removal. *ACM Applied Computing Review*, 13(4):30–42, 2013.

## ABOUT THE AUTHORS:

Dr. Md Rakibul Islam is an Assistant Professor at the Department of Computer Science, University of Wisconsin - Eau Claire, USA. His research interests include human aspects in software engineering, software security, source code analysis, & natural language processing. To solve problems in his research areas, he developed various useful tools and generated interesting insights by applying techniques, such as machine learning, data mining, and graph theories. To disseminate his research findings & outputs, he has published several scholarly articles in reputed journals & conferences. He has also served as a reviewer & co-reviewer to reviewing manuscripts submitted in different journals & conferences. Rakib also has almost seven years of industrial experience.

Dr. Minhaz F. Zibran is an Associate Professor at the Department of Computer Science, University of New Orleans, USA. His research fights software bugs and security vulnerabilities using program code analysis and manipulation while also taking into account the human factors that affect software quality. Minhaz has co-authored many scholarly articles published in ACM and IEEE sponsored international conferences and reputed journals. Minhaz also has experience of working in software industry. He has been actively involved in organizing international conferences (e.g., MSR, ICDF2C, IWSC, SEMotion, ICPC, ICSM, SCAM) and in reviewing manuscripts submitted to reputed journals (e.g., IEEE Software, IEEE Security & Privacy, TOSEM, EMSE, JSS, IST, SQJ).