How Bugs Are Fixed: Exposing Bug-fix Patterns with Edits and Nesting Levels

Md Rakibul Islam
University of New Orleans, USA
mislam3@uno.edu

Minhaz F. Zibran
University of New Orleans, 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 can be exploited in devising techniques for automatic 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 exposes 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**;

KEYWORDS

Program; bug; fix; automatic repair; pattern

ACM Reference Format:
Md Rakibul Islam and Minhaz F. Zibran. 2020. How Bugs Are Fixed: Exposing Bug-fix Patterns with, Edits and Nesting Levels. In *The 35th ACM/SIGAPP Symposium on Applied Computing (SAC '20), March 30-April 3, 2020, Brno, Czech Republic.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3341105.3373880

1 INTRODUCTION

Bug-fixing efforts consume a vast amount of total expenses in software maintenance [6] while nearly 80% of software cost is spent in maintenance [15]. Typical bug-fixing efforts mainly involve two types of tasks: (a) localization of bugs in source code, and (b) bug-fixing edits to source code. A deep understanding of the common bug-fixing patterns can immensely help in minimizing efforts in both of these tasks and also contribute to devising techniques for automated program repair (APR). 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 [22, 23] 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 [22, 33] in the analysis of textual differences among different program entities. However, manual effort is criticized for being error-prone, tedious, incomplete, and imprecise [8, 14]. Recent efforts made use of Abstract Syntax Tree (AST) based code differencing tools (e.g., `ChangeDistiller` [9], `Diff/TS` [13] and `GumTree` [8]) 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* (see examples in Section 2.3). A nested code structure is a hierarchy of AST nodes that 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 [8, 21, 25, 26, 33, 37, 41] 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 bug-fixing editings/changes made in source code and 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 [26, 33, 37].

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 located. These *nesting* patterns 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. Such nested code structure contexts provide opportunities to use that along with other code contexts, such as *textual similarity of code* [36] to locate program faults, and repair those automatically.

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 34 of them confirm those reported in earlier studies.

SAC '20, March 30-April 3, 2020, Brno, Czech Republic
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-6866-7/20/03…$15.00
https://doi.org/10.1145/3341105.3373880

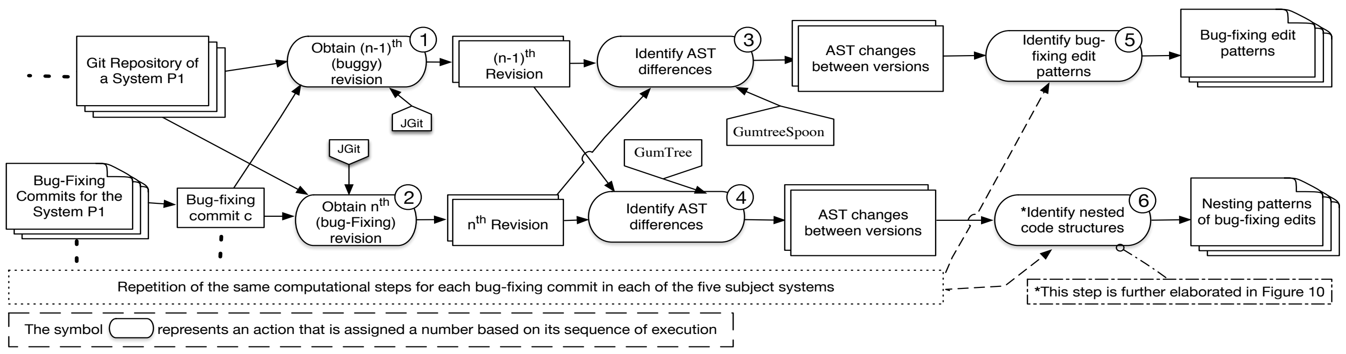

Git Repository of a System P1
Obtain (n-1)th (buggy) revision
1
JGit
(n-1)th Revision
Identify AST differences
3
GumtreeSpoon
AST changes between versions
Identify bug-fixing edit patterns
5
Bug-fixing edit patterns
Bug-Fixing Commits for the System P1
Bug-fixing commit c
JGit
Obtain nth (bug-Fixing) revision
2
nth Revision
GumTree
Identify AST differences
4
AST changes between versions
*Identify nested code structures
6
Nesting patterns of bug-fixing edits
Repetition of the same computational steps for each bug-fixing commit in each of the five subject systems
*This step is further elaborated in Figure 10
The symbol represents an action that is assigned a number based on its sequence of execution

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

- We study locations of bug-fix changes in nested code structures and identify 37 *nesting* patterns that hold the majority of the bug-fix edits. These *nesting* patterns are new (i.e., never targeted before), and add a new dimension in our understanding of bug-fix patterns.

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 procedural steps with necessary discussions.

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 in GitHub. In Table 1, we present the total number of revisions and the number of source lines of code (KLOC) in the last revision for each subject system. We select these five subject systems as these systems have variations in application domains, sizes, number of revisions, and are also used in other studies [17, 35].

Table 1: Subject Systems

Subject System	Application Domain	KLOC (last rev.)	Total # of Revisions	# of Bug-Fixing Rev.
Netty	Network	1,078	8,534	1,103
Presto	SQL	2,869	11,909	841
Facebook-android-SDK	Social Networking	172	671	133
Accumulo	Distributed Key-value store	458	9,734	1,941
Common-maths	Math library	187	6,971	635
Total over all the systems		4,764	37,819	4,653

Moreover, the selected five subject systems can be classified in 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 [36]. Such a combination of selected subject systems provides the opportunity not only to verify existence of earlier detected bug-fixing edit patterns but also to identify new bug-fixing edit patterns if they exist in our dataset.

2.2 Collecting Bug-fixing Commits

For the first three systems, we collect the bug-fixing commits identified by Ray et al. [35]. These bug-fixing commits are distinguished through matching keywords (e.g., bug, defect, issue) in the commit messages and are reported to be 96% accurate [35]. To identify the bug-fixing commits in the remaining last two systems, we use the same keywords used by Ray et al. [35]. The number of bug-fixing revisions for each system is listed in the last column of Table 1.

2.3 Generating AST-differences

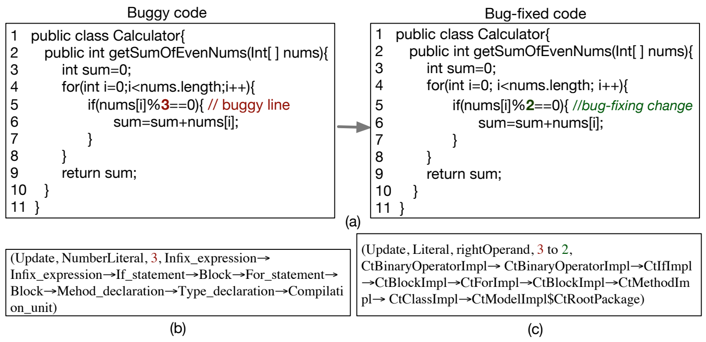
Consider a bug-fixing commit C resulting in the n^{th} revision of a system/project $\mathcal{P}_1$. If a particular line of code $\mathcal{L}$ is modified in the bug-fixing commit C, then it implies that the modification is necessary to fix the bug. Thus, the line of code $\mathcal{L}$ in the $(n-1)^{\text{th}}$ revision is considered as a buggy line. In other words, we consider the changes between the n^{th} and $(n-1)^{\text{th}}$ revisions of $\mathcal{P}_1$ are buggy. Several other studies [16, 29, 34] also adopted the similar approach for distinguishing buggy source code.

At this level, as shown in Figure 1, we obtain the n^{th} and $(n-1)^{\text{th}}$ revisions of $\mathcal{P}_1$ using `JGit` [4]. Then, we capture bug-fixing changes at the AST [8] 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 the background/context that helps in understanding the rest of the content of the paper.

Understanding of `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



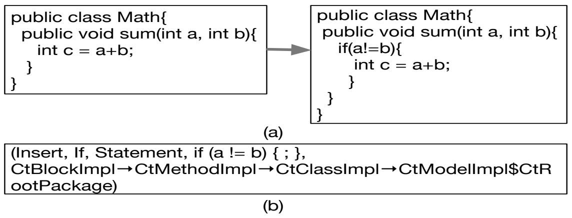
```
1   public class Calculator{
2       public int getSumOfEvenNums(Int[ ] nums){
3           int sum=0;
4           for(int i=0;i<nums.length;i++){
5               if(nums[i]%3==0){ // buggy line
6                   sum=sum+nums[i];
7               }
8           }
9           return sum;
10      }
11  }
```

```
1   public class Calculator{
2       public int getSumOfEvenNums(Int[ ] nums){
3           int sum=0;
4           for(int i=0; i<nums.length; i++){
5               if(nums[i]%2==0){ //bug-fixing change
6                   sum=sum+nums[i];
7               }
8           }
9           return sum;
10      }
11  }
```

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

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

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`



```
public class Math{
  public void sum(int a, int b){
    int c = a+b;
  }
}
```

```
public class Math{
  public void sum(int a, int b){
    if(a!=b){
      int c = a+b;
    }
  }
}
```

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

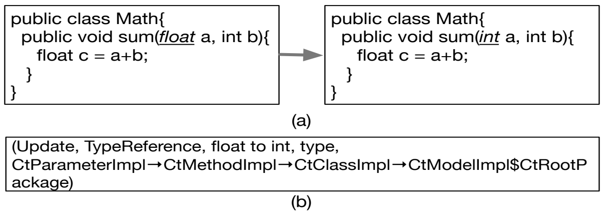
Figure 3: (a) Adding an `if` statement as a precondition to fix a bug and (b) corresponding representation of the bug-fixing change using `GumtreeSpoon`

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, 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.

Understanding of `GumtreeSpoon`'s output. While the outputs of `GumtreeSpoon` are almost similar to the outputs of `GumTree` in terms of presentation, 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)



```
public class Math{
  public void sum(float a, int b){
    float c = a+b;
  }
}
```

```
public class Math{
  public void sum(int a, int b){
    float c = a+b;
  }
}
```

(Update, TypeReference, float to int, type, CtParameterImpl→CtMethodImpl→CtClassImpl→CtModelImpl$CtRootPackage)

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`

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`.

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 (e.g., `conditional operator` and variables (i.e., a and b) additions). 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).

Reason to use two different tools to answer RQs. While the 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.

3 CAPTURING BUG-FIXING 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. [33]. 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. [33]. 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 Making Sense of `GumtreeSpoon`'s Output

Here, we manipulate the `GumtreeSpoon`'s outputs 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 patterns that we identify using Algorithm 1 are termed as *GSPatterns* in the rest of this paper.

3.2 Mapping GSPatterns to PanPatterns

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) [33]. However, we have to leverage the attribute *modified source code* to 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.

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. 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 semicolon, we also check whether 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 an inserted `if` is added as postcondition if that is not a precondition.

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;
```

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

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

Table 2: Distributions of identified *PanPatterns*

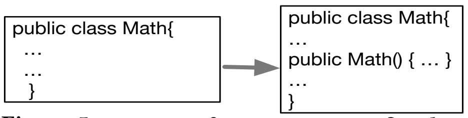
Category	Pattern Name	# (%)	Total (%)
Method Declaration (MD)	Change of method declaration (MD-CHG)	4,116 (17.54%)	7,687 (33%)
	Addition of a method declaration (MD-ADD)	1,916 (8.17%)	
	Removal of a method declaration (MD-RMV)	1,655 (7.05%)	
If-related (IF)	Addition of post-condition check (IF-APTC)	1,975 (8.42%)	4,877 (21%)
	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%)
	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%)
	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 (1%)	1,401 (1%)
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	# (%)	Total (%)
Local Variable (LV)	Update impl. of `local variable` (LV-IMPL)	4,043 (15.41%)	7,709 (29%)
	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 (MC-D)	1,896 (7.23%)	
Return (RT)	Update impl. of `return` stat. (RT-IMPL)	3,361 (12.81%)	4,200 (16%)
	Addition or deletion of `return` stat. (RT-AD)	839 (3.20%)	
Assignment (AS)	Addition or deletion of `assignment block` stat. (AS-AD)	3,390 (12.92%)	3,390 (12.92%)
Constructor (CT)	Addition or deletion of `constructor` (CT-AD)	578 (2.20%)	1,013 (4%)
	Parameter update in `constructor` (CT-Param)	435 (1.66%)	
Throw (TW)	Update of impl. of `throw` stat. (TW-IMPL)	651 (2.42%)	861 (3.28%)
	Addition or deletion of `throw` stat. (TW-AD)	210 (0.80%)	
Class or Interface (CI)	Addition or deletion of class or interface (CI-AD)	480 (2%)	480 (2%)
Wrap or 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` stat. (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` stat. (EN-AD)	112 (0.43%)	112 (0.43%)
***impl.=implementation; stat.=statement**		**Overall total**	**26,241 (100%)**

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.

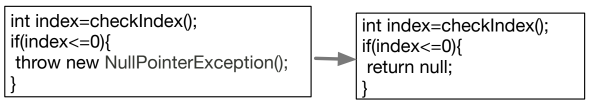
The pattern MD-CHG experiences the highest number of bug-fixing changes (17.54%) followed by the patterns MC-DNP (10.24%) and IF-APTC (8.42%). Interestingly, those three patterns are from three distinct categories. MD-ADD, CF-CHG, and MD-RMV are the next three patterns that experience the highest number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



```
public class Math{
  ...
  ...
  }
```

```
public class Math{
...
public Math() { ... }
...
}
```

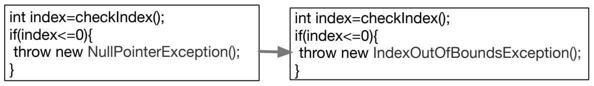
Figure 5: Insertion of a `constructor` to fix a bug



```
int index=checkIndex();
if(index<=0){
 throw new NullPointerException();
}
```

```
int index=checkIndex();
if(index<=0){
 return null;
}
```

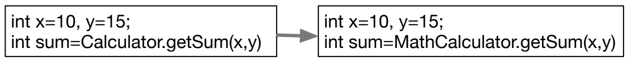
Figure 6: Deletion of a `throw` statement to fix a bug



```
int index=checkIndex();
if(index<=0){
 throw new NullPointerException();
}
```

```
int index=checkIndex();
if(index<=0){
 throw new IndexOutOfBoundsException();
}
```

Figure 7: Modification of a `throw` statement to fix a bug



```
int x=10, y=15;
int sum=Calculator.getSum(x,y)
```

```
int x=10, y=15;
int sum=MathCalculator.getSum(x,y)
```

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

Except 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%.

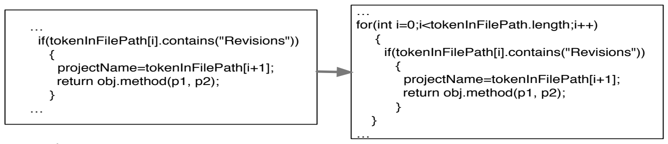
3.3.2 *New bug-fixing edit patterns.*

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 [25, 26, 37], we identify completely four new bug-fixing edit patterns as indicated in Table 3. In the following, we briefly define the new patterns, which are relatively complex, while some of those patterns can be easily understood from their names such as addition or deletion of a method call (MC-AD) or a class/interface (CI-AD).

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



```
...
if(tokenInFilePath[i].contains("Revisions"))
  {
    projectName=tokenInFilePath[i+1];
    return obj.method(p1, p2);
  }
...
```

```
...
for(int i=0;i<tokenInFilePath.length;i++)
  {
    if(tokenInFilePath[i].contains("Revisions"))
      {
        projectName=tokenInFilePath[i+1];
        return obj.method(p1, p2);
      }
  }
...
```

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

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 where the class of the method `getSum` is changed to fix a bug.

Parameter update in Constructor (CT-Param). Similar to pattern MD-CHG, parameters of a `constructor` can be changed 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.

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 equal amount of bug-fixing changes (≈13%). 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 Pattern Mining of Nested Code 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 parents 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 value 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.

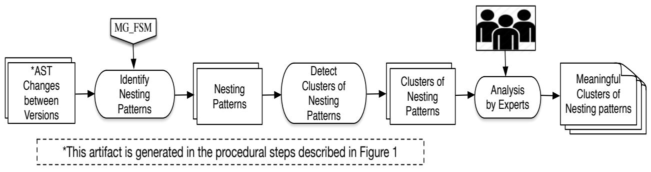

MG_FSM
*AST Changes between Versions
Identify Nesting Patterns
Nesting Patterns
Detect Clusters of Nesting Patterns
Clusters of Nesting Patterns
Analysis by Experts
Meaningful Clusters of Nesting patterns
*This artifact is generated in the procedural steps described in Figure 1

Figure 10: Steps to identify the nesting patterns

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* [11, 40], where sub-patterns that are contained within longer patterns are ignored. As we aim to identify nesting patterns of bug-fixing changes, we find the maximal pattern mining is preferable in our case.

While there are few commonly used sequential pattern mining algorithms available [10], we use recently proposed `MG-FSM` algorithm [27] 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 in parallel running using map-reduce (Hadoop) functionality. 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 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 characterize and label the identified nesting patterns manually by experts where developers commonly perform bug-fixing changes.

To cluster nesting patterns, we use k-medoids [19] algorithm that is a variant of k-means [18] algorithm. k-medoids is known for more robustness against noises and outliers compared to k-means [19]. To determine the optimal number of clusters (i.e., k in k-medoids), we use *gap statistic* [38] method over other available options, such as *elbow* and *silhouette* methods. We use *gap statistic* method as it can be applied to any clustering method (i.e., k-medoids, k-means, and hierarchical clustering). Using the gap statistic, we find the number of optimal clusters is 10 for our data.

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 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 between S_1 and S_2.

Using the package `python-string-similarity` [3], we compute the LCS metric values. Then, to cluster nesting patterns, we run the open source implementation of k-medoids algorithm implemented in the package `Pycluster 1.49` [2].

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 and characterized by human experts to gain meaningful insights of the nesting patterns' clusters.

Table 4: Dominant *Nesting* Patterns of Bug-Fixing Edits

Category	**ID, Mined Nesting Patterns**	# (%)	**Total** (%)
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 (04.08%)	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 (05.93%)	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%)**

4.3 Characterization of the Clusters

Using subjective evaluation, each author separately characterizes the nesting patterns in each cluster in terms of their nodes' hierarchies. By observing nodes' hierarchies of the patterns, two sets of nodes are created: (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 label node's block from l is contained in a high-level node block from h. Such an 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 second column of the last row in Table 4).

Cohen's kappa coefficient κ [7] is used to measure agreement between two authors in charatcterizing patterns. κ value 0.79 indicates high-level agreement on the characterization of the patterns of the clusters. For each disagreement, 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 ended with `Method_declaration→Type_declaration→Compilation_unit`, we truncate that for better 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 that 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-levels 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 containment.

5 THREATS TO VALIDITY

Construct Validity. For the first three projects, we use the bug-fixing commits that are identified by Ray et al. [35]. To distinguish those bug-fixing commits, they used a technique similar to the approach of Mockus and Votta [28]. The same approach is used for detecting bug-fixing commits of the last two projects. The accuracy of this approach may be questioned. However, this approach was reported 96% accurate [35].

To detect bug-fixing edit patterns, we have considered only nodes found before the occurrence of the first block node (if available) in a NCS. Someone may be skeptical in 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).

In capturing the 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.

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 [8]. Moreover, `ChangeDistiller` works at the statement level, preventing the detection of certain fine-grain patterns.

Similar to `GumtreeSpoon`, there is another tool `ClDiff` [14] 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 [33]. The library `JGit` [4] is used in our work to extract a buggy and its previous revisions. This library was also applied for similar purposes in other studies [12, 31].

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 be generalizable for industrial systems and source code written in languages other than Java.

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. Therefore, it should be possible to replicate the study.

6 RELATED WORK

Pan et al. [33] 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. [41] to identify 11 bug-fixing edit patterns, however, they used 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 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 [8].

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 [21, 25, 26, 33, 37] carried out to identify bug-fixing edit patterns till date. In addition, we have identified four new patterns in Constructor, Catch and Enum categories (see Table 3).

Kim et al. [21] 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 `Kenyon` framework [5] and clustered those patches using `groums` [30] to minimize human efforts. Again, Sobreira et al. [37] manually analyzed only 395 patches collected from *Defects4J* [20] 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 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. [39] 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 can answer this question as ‘yes’ by observing the 21st and 22nd clusters in the SYN category presented in Table 4.

To overcome the problems of manual approaches, automatic techniques are developed to identify bug-fixing edit patterns. Martinez and Monperrus [26] identified 20 bug-fixing edit patterns using the AST based code differencing tool `ChangeDistiller` [9]. In our study, we have used the tool `GumtreeSpoon` [1] developed based on `GumTree` [8], which is more accurate than `ChangeDistiller`.

Few other studies [24, 32] identified fixing patterns of violations of static coding principles. However, in our study, we have studied real bug-fixing changes instead of coding principles’ violations.

7 CONCLUSION

In this paper, we have reported 38 bug-fixing *edit* patterns, which is the highest number of bug-fixing *edit* patterns identified in a single study. Moreover, we have discovered four new bug-fixing *edit* patterns. The rest 34 identified bug-fixing *edit* patterns confirm those reported earlier.

Using sequential pattern mining and clustering techniques, we have also exposed 37 new bug-fixing *nesting* patterns, which capture the locations of the bug-fixing edits within the nested code structure surrounding them. These new set of *nesting* patterns is a novel contribution that adds a new dimension to our understanding of bug-fixing patterns.

Our analysis of the *nesting* patterns reveals additional insights into bug-fix patterns. We have found that any nodes/blocks associated with `if` blocks are the most bug-prone. The *nesting* pattern “`if` block inside `loop` block” experience 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 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` are bug-prone.

The findings from this work are derived from both quantitative and qualitative analyses that 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 bug and generating patches for those bugs successfully. Our future work will explore these possibilities.

ACKNOWLEDGEMENT

This work is supported in part by the SCoRe grant at the University of New Orleans.

REFERENCES

[1] verified : Sep 2019. *GumtreeSpoon - Spoon version of GumTree*. https://github.com/SpoonLabs/gumtree-spoon-ast-diff.

[2] verified : Sep 2019. *PyCluster - Clustering module for Python*. https://bioconda.github.io/recipes/pycluster/README.html.

[3] verified : Sep 2019. *Python-String-Similarity*. https://github.com/luozhouyang/python-string-similarity/blob/master/README.md.

[4] verified: Sep 2018. *JGit*. https://www.eclipse.org/jgit/.

[5] J. Bevan, E. Whitehead, S. Kim, and M. Godfrey. 2005. Facilitating software evolution research with kenyon. In *FSE*. 177–186.

[6] E. Campos and M. Maia. 2017. Common Bug-fix Patterns: A Large-Scale Observational Study. In *ESEM*. 404–413.

[7] A. Cantor. 1996. Sample-size calculations for Cohen’s kapp. *Psychological Methods* 1, 2 (1996), 150–153.

[8] J. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus. 2014. Fine-grained and Accurate Source Code Differencing. In *ASE*. 313–324.

[9] B. Fluri, M. Wursch, M. Pinzger, and H. Gall. 2007. Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction. *IEEE Transactions on Software Engineering* 33, 11 (2007), 725–743.

[10] P. Fournier-Viger, J. Lin, A. Gomariz, T. Gueniche, A. Soltani, and Z. Deng. 2016. The SPMF Open-Source Data Mining Library Version 2. In *MLKDD*. 36–40.

[11] P. Fournier-Viger, C. Wu, A. Gomariz, and V. Tseng. 2014. VMSP: Efficient Vertical Mining of Maximal Sequential Patterns. *Advances in Artificial Intelligence* 8436 (2014), 83–94.

[12] G. Greene and B. Fischer. 2016. CVExplorer: Identifying Candidate Developers by Mining and Exploring Their Open Source Contributions. In *ASE*. 804–809.

[13] M. Hashimoto and A. Mori. 2008. Diff/TS: A Tool for Fine-Grained Structural Change Analysis. In *WCRE*. 279–288.

[14] K. Huang, B. Chen, X. Peng, D. Zhou, Y. Wang, Y. Liu, and W. Zhao. 2018. ClDiff: Generating Concise Linked Code Differences. In *ASE*. 679–690.

[15] Research T. Institute. 2002. *The Economic Impacts of Inadequate Infrastructure of Software Testing*. RTI Project Report 7007.011. National Inst. of Standards and Tech.

[16] J. Islam, M. Mondal, and C. Roy. 2016. Bug Replication in Code Clones: An Empirical Study. In *SANER*. 68–78.

[17] M. Islam and M. Zibran. 2018. On the Characteristics of Buggy Code Clones: A Code Quality Perspective. In *IWSC*. 23 – 29.

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

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

[20] R. Just, D. Jalali, and M. Ernst. 2014. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *ISSTA*. 437–440.

[21] D. Kim, J. Nam, J. Song, and S. Kim. 2013. Automatic patch generation learned from human-written patches. In *SEKE*. 802–811.

[22] M. Kim and D. Notkin. 2009. Discovering and representing systematic code changes. In *SEKE*. 309–319.

[23] S. Kim, K. Pan, and E. Whitehead. 2006. Memories of bug fixes. In *FSE*. 35–45.

[24] K. Liu, D. Kim, T. Bissyande, S. Yoo, and Y. Traon. 2018. Mining Fix Patterns for FindBugs Violations. *IEEE Transactions on Software Engineering* (2018), 1–24.

[25] M. Martinez, L. Duchien, and M. Monperrus. 2013. Automatically Extracting Instances of Code Change Patterns with AST Analysis. In *ICSME*. 22 – 28.

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

[27] I. Miliaraki, K. Berberich, R. Gemulla, and S. Zoupanos. 2013. Mind the Gap: Large-Scale Frequent Sequence Mining. In *CMD*. 797–808.

[28] A. Mockus and L. Votta. 2000. Identifying reasons for software changes using historic databases. In *ICSME*. 120–130.

[29] M. Mondal, C. K. Roy, and K. A. Schneider. 2017. Identifying Code Clones having High Possibilities of Containing Bugs. In *ICPC*. 99–109.

[30] T. Nguyen, H. Nguyen, N. Pham, J. Al-Kofahi, and T. Nguyen. 2009. Graph-based mining of multiple object usage patterns. In *FSE*. 383–392.

[31] H. Osman, M. Lungu, and O. Nierstrasz. 2014. Mining Frequent Bug-Fix Code Changes. In *SANER*. 343–347.

[32] H. Oumarou, N. Anquetil, A. Etien, S. Ducasse, and K. Taiwe. 2015. Identifying the exact fixing actions of static rule violation. In *SANER*. 371–379.

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

[34] F. Rahman, C. Bird, and P. Devanbu. 2010. Clones: what is that smell?. In *MSR*. 72–81.

[35] B. Ray, V. Hellendoorn, S. Godhane, Z. Tu, A. Bacchelli, and P. Devanbu. 2016. On the "Naturalness" of Buggy Code. In *ICSE*. 428–439.

[36] R. Saha, Y. Lyu, H. Yoshida, and M. Prasad. 2017. ELIXIR: Effective Object-Oriented Program Repair. In *ASE*. 648–659.

[37] V. Sobreira, T. Durieux, F. Madeiral, M. Monperrus, and M. Maia. 2018. Dissection of a Bug Dataset: Anatomy of 395 Patches from Defects4J. In *SANER*.

[38] R. Tibshirani, G. Walther, and T. Hastie. 2001. Estimating the Number of Clusters in a Data Set Via the Gap Statistic. *Journal of the Royal Statistical Society* 63, 2 (2001), 411–423.

[39] M. Tufano, J. Pantiuchina, C. Watson, G. Bavota, and D. Poshyvanyk. 2019. On Learning Meaningful Code Changes via Neural Machine Translation. In *ICSE*. 25–36.

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

[41] R. Yue, N. Meng, and Q. Wang. 2017. A Characterization Study of Repeated Bug Fixes. In *ICSME*. 422–432.