IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Date of publication xxxx 00, 0000, date of current version xxxx 00, 0000.

Digital Object Identifier 10.1109/ACCESS.2017.DOI

History-driven Fix for Code Quality Issues

JIANGTAO XUE, XINJUN MAO, YAO LU, YUE YU AND SHANGWEN WANG

Key Laboratory of Software Engineering for Complex Systems, College of Computer Science, National University of Defense Technology, Changsha 471003, China

Corresponding author: Xinjun Mao (xjmao@nudt.edu.cn).

This work was supported by National Key Research and Development Program of China under Grant No: 2018YFB1004202, and Laboratory of Software Engineering for Complex Systems.

ABSTRACT To ensure the internal code quality of contributions in open source software (OSS) communities, static analysis tools (e.g. Code Climate and SonarQube) have been integrated into the modern pull-based workflow for detecting code quality issues (CQIs). Automated CQI fixing is conducive to improve the efficiency of converging massive contributions. In this paper, we propose a history-driven approach to automatically fix CQIs utilizing the fixing knowledge mined from the change history in the code repositories. We collected 5,047,678 CQI isstances, 31,013 fixes of the CQIs that were detected by SonarQube from 206 GitHub projects and mined 68 common Fix Patterns for 56 CQI types. Evaluated by fixing the unfix CQI Instances, we find that the average correctness of our approach is 80% in top 1 fix patch and 69.17% of Top 5. We further conducted a live study by sending CQI's fix patches to GitHub projects. The results show that developers approved or merged 11 of 17 patches.

INDEX TERMS Open Source, Code Quality, Mining GitHub Repository, Fix Pattern

I. INTRODUCTION

The code quality plays an essential role for software project's success [5]. Generally, code quality can be classified as internal and external quality attributes [4]. The external quality attributes are reflected at runtime stage such as functionality and correctness, while internal quality attributes exist in development and maintenance [2], which is concerned by developers, such as maintainability and readability. High internal quality source code not only improves the readability of code and help developers quickly understand explicit and implicit meanings expressed by code, but also reduces the risk of crash and the cost of software maintenance [7]. In this paper, we focus on the internal quality attributes from the point of view of developers, since it's critical for developers to review, maintain source code in Pull-based code contributions of OSS community. Code Quality Issue (CQI) can be detected by static analysis tools, such as Code Climate and SonarQube, which have been wildly utilized in OSS community. CQIs can be classified as Vulnerability, Bug and Code Smell.

A previous study found that 30% of the code contributions comes from 70% of the contributors in a OSS project, and these 70% contributors, whose development experience ranges from novice to experienced, called casual contributors, casually submit a very limited number of violation fixes and feature enhancements [2]. According to The State of the Octoverse 2018 [8], the GitHub community reached 31 million developers working across 96 million repositories. There have been 1 billion public commits, 12.5 million active issues and 47 million pull requests (PRs) since September 2017. Ensuring and controlling code quality of Pull-based code contributions made by casual contributors has been more and more challenging because of the rapid development of OSS community.

As a result, Continuous Inspection for Pull-based contributions are required in OSS community, which automatically utilizes static analysis tool to detect CQIs. Continuous Inspection provides continuous code quality management that incorporates shorter feedback loops to ensure the rapid resolution of quality issues [2]. Nowadays developers usually manually fix CQI when executing Continuous Inspection, including understanding CQI Rule, searching for proper fix solution, and revising source code which is very time-consuming. As a result, it is urgently necessary to propose an automated approach of fix CQIs in OSS community to improve the efficiency of converging massive contributions.

What's more, there exists tremendous CQI Fix Instances in the code change history of OSS repositories. Figure

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2019.2934975, IEEE Access


```
        } else {
            table.addCell("-");
        }
        final Map<String, String> serviceAttributes = info.getServiceAttributes();
```

FIGURE 1. CQI example taken from $RestNodesAction.java$ in project $elasticsearch$

```
-       final Map<String, String> serviceAttributes = info.getServiceAttributes();
+       final Map<String, String> serviceAttributes = info == null ? null : info.getServiceAttributes();
        if (serviceAttributes != null) {
                table.addCell(serviceAttributes.getOrDefault("http_address", "-"));
        }
```

FIGURE 2. CQI fix instance taken from commit $a70be3$ of $elasticsearch$ project

1 shows a CQI instance of bug severity, which means $NullPointerException$ might be thrown as variable $info$ is nullable here. CQI Rule is that reference to null should never be dereferenced or accessed. Doing so will cause a NullPointerException to be thrown. At best, such an exception will cause abrupt program termination. At worst, it could expose debugging information that would be useful to an attacker, or it could allow an attacker to bypass security measures. Figure 2 shows the fix instance taken from commit $a70be3$ of $elasticsearch$ project by adding Ternary Conditional Operator to ensure info in not null. Kui Liu et al. did a study about violations detected by FindBugs in OSS Community and found that violations types are recurrent. They noted that about 12% violation types account for 81.4% of all distinct violations [9] and they can be fixed by similar code change operations.

In this paper, we propose a history-driven approach to automatically fix CQIs. Some IDEs, e.g. Eclipse, can supply "$QuickFix$" [6] to automatic fix some program warnings and errors while typing and compiling with pre-defined templates. Different with "$QuickFix$", our approach focus on mining diversified, real-world Fix Patterns from code change history and apply them to fix CQIs detected by SonarQube. As far as we know, our approach is the first attempt to automatically fix CQIs using history-driven approach. On the one hand, our approach can improve the efficiency of converging massive contributions in OSS community. On the other hand, by utilizing fix knowledge (so-called Fix Pattern) learned from a great amount of code change history in OSS Community (e.g. GitHub), our approach can supply diversified, real-world fix patterns for CQI to be fixed.

The contributions of our work are as follows:

1) We proposed a history-driven approach to automatically fix CQIs based on mining CQI fix pattern in software code change history, matching CQI fix pattern with code context and histories data and generate ranked fix patch list. Our approach is the first attempt to automatically fix CQIs using history-driven approach.
2) We collected 5,047,678 CQI Instance detected by SonarQube and 31,013 fixed CQIs. We found that 7.12% CQI types account for more than 90% of all CQI Instances in 206 open source project. Despite some false positives, our study show that fixed CQI types account for 73.7% of all detected CQI types. Our study show that CQI is recurrent and most of CQI types can be fixed by developers.
3) We mined and abstracted 68 fix patterns of 57 CQI types utilizing our history-driven approach. The results show that the average correctness of fix pattern is 80% in top 1 fix patch and 69.17% of Top 5 fix patterns can be applied to fix CQI Instance. We also conduct a live study in GitHub and find that developers approved or merged 11 of 17 Pull Requests.

The rest of paper is organized as follows. Section II discusses related work. Section III details our approach. Section IV show the experiment results and evaluations of the experiment results. Section V identifies the threat to validity of our study. Finally, Section VI concludes the paper and discusses the future work.

II. RELATED WORKS

A. STATIC ANALYSIS AND CODE QUALITY

Nowadays Static Analysis has been untilized to assess and maintain Code Quality in OSS community. Avgustinov et al. [33] presented an approach for tracking static analysis violations over the revision history. They conclude that these fingerprints are well-defined and capture the individual developers coding habits by performing an experimental study on several lager open-source projects. Nagappan et al. [3] investigated the use of automated inspection for industrial software system at Nortel Networks. They proposed a defect classification scheme in order to enumerate the types of defects that can be identified by static tools, and concluded that automated code inspection faults can be used as efficient

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2019.2934975, IEEE Access

predictors of failures. Yao Lu et al. investigated internal quality of the code made by casual Contributors in OSS projects within GitHub [2]. They proposed a method to estimate developers' code quality using the "Code Quality Issue Density (CQID)" metric and concluded that casual contributors tend to introduce a greater quantity of CQIs with greater severity than the main contributors and casual contributors tend to introduce CQIs on coding convention, errorhandling, CWE (Common Weekness Enumeration) and brain-overload, which can decline the readability, reliability, efficiency and testability of the software.

Based on previous study about static analysis vioation, correlation between static analysis and code quality, and the relaitionship between developers and code quality in OSS community, we focus on the distributions of CQIs and the fix of them in OSS Community. We emphasize that automatically fix of CQI should be untilized in OSS Community to improve the efficiency of converging massive contributions in OSS community.

B. PROGRAM REPAIR

Generally, there are mainly two lines about the program repair research [9]: (1) the patch hint suggestion and (2) fully automated repair. The patch hint techniques suggest code fragments that could assist users to create a patch without human-intervention [9] while the fully automated repair focus on automatically generating patches which can be integrated into program. There have been several studies about program repair in [14]. Automated program repair is pioneered by Gen-Prog [12], [13]. There are other notable approaches including contract-based fixing [15], conditional statement repair [16], and behavior-based program repair [17].

There are two main classes of program repair approaches : the generate-and-validate approach and the semantics-driven approach [14]. In terms of the generate-and-validate approach, in addition to GenProg [12], Marriagent [22] and SCRepair [23], we introduce some history-driven methods. Koyuncu et al. proposed FixMiner that explores the hierarchical AST data include contextual actual code changes. They implemented FixMiner as part of automated program repair system. The main limitation of FixMiner is that it extracts the fix pattern of a single repair action(either UPD, ADD, DEL, MOV in all AST nodes rather than the combination of them) [24]. Gupta et al. proposed DeepFix [25] to fix common C programming errors. DeepFix adopted an end-to-end solution based on deep learning and can successfully fix many error types in students' programs in an actual programming course. Xuan-Bach and David [10] presented a novel technique for history-based program repair. Using a large repository of real patches, the method evaluates the fitness or suitability of a candidate patch by assessing the similarity between the patch and the prior bug-fixing patches. They demonstrated that their method is effective in fixing 23 bug types correctly. In terms of semantics-driven approaches, we introduce the SemFix and DirectFix. SemFix [26] synthesizes that fixes consist of a single changed statement. Thus the fixes that require changes in multiple locations of the code are out of the scope of this technique. DirectFix [27] repairs faulty expressions inside a program with a monolithic approach. Previous study focus on exploring effective automated program repair approach, but they only consider the exposed program bugs, which are code external quality attributes and make test case fail during runtime. We focus on exploring history-driven approach to mine fix pattern of CQIs, which are code internal quality attributes, and apply them to automated fix CQIs.

C. CODE CHANGE HISTORY MINING

Code repositories contain knowledge that can be used to discriminate between good and bad source code. So far, the knowledge available in source code repositories has not yet been fully leveraged. Kim et al. [28] presented a bug finding algorithm using bug fix memories, which contain a project-specific bug and fix knowledge base that was developed by analyzing the history of bug fixes. Martinez et al. [29] gave empirical results on the nature of human bug fixes at a large scale and a fine granularity with abstract syntax tree(AST) differencing by analyzing thousands of bug fix transactions of software repositories. Based on the insight, PAR [30] leveraged common fix patterns for automated program repair. Zhong and Su [31] conducted a study on fixing changes in OSS projects. Tan et al. [32] analyzed anti-patterns that may interfere with the process of automated program repair.

III. APPROACH

A. OVERVIEW

In this section, we give an overview of our approach. The overall goal of our approach is to mine fix patterns for CQIs and generate CQI fix patches. To achieve that goal, our approach involves 4 main phases(described in Figure 3 and Figure 4): CQI Fix Instance Collecting, Fix Pattern Mining, Fix Pattern Matching and Fix Patch Recommending, and each phase consists of several steps. First, we analyze the OSS repositories and collect the CQI Fix instances using SonarQube. We extract CQI fixing commits and collect CQI source code file in both the pre-fix version and post-fix version. Second we mine Fix Patterns using GumTree [30]. After that we build library of CQI Fix Patterns.

Third we matche the CQI Fix Patterns for an incoming CQI. The input is the Source File of the CQI and the of Fix Pattern library. The matching algorithm ranks the matched Fix Patterns by Code Context similarity. The output is a ranked list of Fix Patterns. Finally, our approach modifies the CQI Source File guided by the matched Fix Patterns and generates a list of fix patches. The developers can select the most suitable patch to fix CQI.

B. CQI FIX INSTANCE COLLECTING

In this phase, we collect the CQI fix history from OSS community. We gather popular Java projects from GitHub. We use GitHub API, with which we can access GitHub

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2019.2934975, IEEE Access

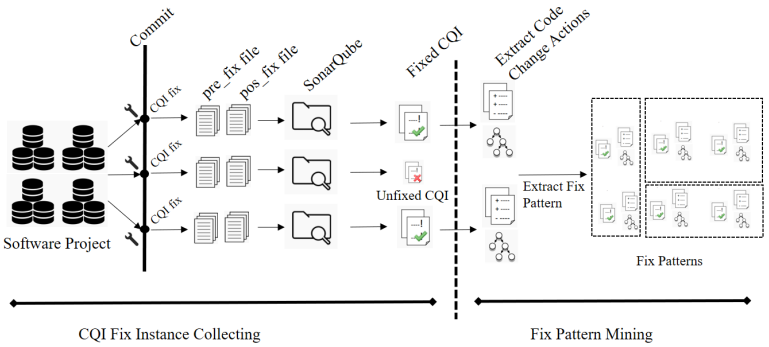


FIGURE 3. The process of CQI Fix Instance Collecting and Fix Pattern Mining

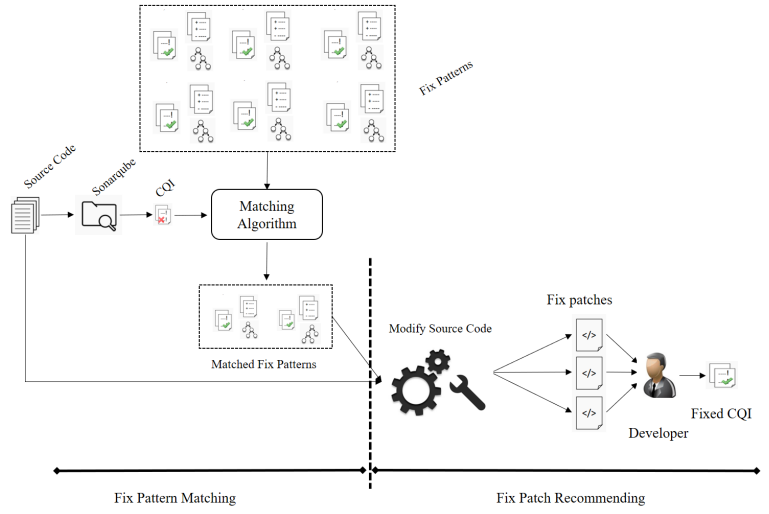


FIGURE 4. The process of Fix Pattern Matching and Fix patches Recommending

latest data, e.g. pull request, commit, repository, etc, to filter the 300 most popular java projects sorted by stars which corresponds to the number of GitHub users that have expressed interest in that project. For each project, we use the $gitlog$ command. By default, with no arguments, git log command lists the commits made in curent branch of the repository in reverse chronological order, that is, the most recent commits show up first. Then we use $gitlog$ with the $-grep$ argument, that can search for keywords in the commit messages, to filter CQI fix commit. We deem a commit a CQI fix if it simultaneously satisfies that there are the keywords such as $fix, CQIfix, codeflaw, codequality, checkstyle$, $findbugs, sonarqube$ in its commit message.

Once we get a CQI fix commit, we set the repository to the previous version and use SonarQube to analyze the source code, which is a powerful static analysis tool and supports more than 20 programming languages and covers 7 axes of code quality: architecture & design, duplications, unit tests, complexity, potential bugs, coding rules and comments. After that, we reset the repository to the fix version and use SonarQube again with same key. SonarQube can record every CQI's status in the same scan task(project key), so if there are some CQI fixed by developers, the status of these CQI will be $closed$, and the resolution of them are $fixed$.

After analyzing both the pre-fix version and the post-fix version with SonarQube, we collect all the CQIs whose resolution are fixed and extract the fix hunks in the fix commit using Algorithm 1.

The pseudocode of our approach to extract the fix hooks from the change history is shown in Algorithm 1. There are several change regions in the git logfile and we define the specific change region around the CQI code as fix hook. It takes as its input the logfile storing the change history, the commit SHA of the post-fix version, and the CQI line. Our algorithm first locates the fix hook by matching the commit SHAs and the changed file name. Then, for each single code in the fix hook, the algorithm computes its line number in the pre-fix version and post-fix version. The algorithm extracts each

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2019.2934975, IEEE Access


```
CQI:Equality tests should not be made with floating point values.

Issue_line:324

if (top.current.key != key) {

fix_hunk:
+++ b/core/src/main/java/org/elasticsearch/search/aggregations/bucket/histogram/InternalHistogram.java
@@ -321,8 +321,9 @@ public final class InternalHistogram extends InternalMultiBucketAggregation<Inte
            do {
                final IteratorAndCurrent top = pq.top();
-               if (top.current.key != key) {
-                // the key changes, reduce what we already buffered and reset the buffer for current buckets
+               if (Double.compare(top.current.key, key) != 0) {
+                    // The key changes, reduce what we already buffered and reset the buffer for current buckets.
+                    // Using Double.compare instead of != to handle NaN correctly.
                    final Bucket reduced = currentBuckets.get(0).reduce(currentBuckets, reduceContext);
                    if (reduced.getDocCount() >= minDocCount || reduceContext.isFinalReduce() == false) {
                    reducedBuckets.add(reduced);
```

FIGURE 5. The Fix Instance collected from commit $16431a$ of $elasticsearch$ project

Algorithm 1: Algorithm of Extracting Fix Hook

Input: logfile, commit_SHA, issue_line
Output: fix hooks

```
1  locate the fix hook in the logfile;
2  line = logfile.read_line();
3  Compute pre_line_num;
4  Compute post_line_num;
5  Initialize hooks ;
6  while not at end of fix hook do
7      Initialize hook ;
8      Compute the pre_file_cursor;
9      Compute the post_file_cursor;
10     if line is added then
11         hook.line_num = post_file_cursor;
12         hook.line = line ;
13         hook.add_line_num = pre_file_cursor;
14     end
15     if line is deleted then
16         hook.line_num = pre_file_cursor;
17         hook.line = line ;
18     end
19     if hook is not NULL then
20         apppend hook to hooks;
21     end
22     line=logfile.read_line()
23 end
24 return hooks ;
```

single changed line of code, stores it in the hook and returns the set of hooks. The contents of the added line and deleted line are shown in Lines 10-18. The difference between them is that the added line should record the line number of the post-fix version while the deleted line does not. Figure 5 shows a fix instance taken from InternalHistogram.java in commit 16431a of elasticsearch project.

C. FIX PATTERN MINING

1) CQI Tokenization

CQIs detected by SonarQube report the issue codes from issue start to issue end. Code in issue lines usually contain noise (e.g. the various variable names). These meaningless information could challenge the process of extracting common fix pattern for CQI, so we do CQI tokenization to abstract the CQI code both on source level and AST level.

Firstly, CQI issue codes are parsed into AST and tokenized into textual vectors with deep-first search algorithm to obtain the AST nodes sequence. Then we filter out the noisy AST nodes (e.g. type is $SimpleName$) , and abstract its label. For example, the code "$equals(Objecto)\{$" is tokenized as a textual vector of 9 tokens ($SimpleName\ equalsSingleVariableDeclaration\ SimpleType\ ObjectSimpleNameObjectSimpleNameo$). Then all the variable names are renamed as 'V'. If there are more than 1 variable, we simply name them 'V', 'V0', 'V1', etc. As a result, we change code "$equals(Objecto)\{$" into "$equals(ObjectV)\{$", and the textual vector of AST nodes ($SimpleNameequals\ SingleVariableDeclaration\ SimpleTypeObjectSimpleNameObjectSimpleNameV$). For the constant, we just ignore the concrete value (e.g. String "$test$" is abstracted to StringLitera).

2) CQI Fix Pattern Mining

Defining Code Change Action

After CQI tokenization, we extract the code change actions between pre-fix version and fix version of CQI source file. We use GumTree [30], a state-of-the-art tree differencing tool that analyzes the AST-level code change actions between 2 source file. Base on the GumTree, we define 4 code change action and its format:

- UPD action. The AST node $node_{src}$ in pre-fix code is replaced by the $node_{dst}$ in fix code. UPD action is formatted as (UPD $node_{src}$ to $node_{dst}$).
- MOV action. The AST node $node_{src}$ in pre-fix code is moved in fix code. MOV action is formatted as (MOV

$node_{src}$).

- INS action. The AST node $node_{dst}$ is added in fix code. INS action is formatted as (INS $node_{dst}$).
- DEL action. The AST node $node_{src}$ in pre-fix code is removed in fix code. DEL action is formatted as (DEL $node_{src}$).

Extracting Merged Deletion Action

When generating the DEL action for subtree of AST that contains more than one node, GumTree starts to remove the leaf nodes and then remove parent nodes which become leaf node after previous DEL action. For example, Figure 6 (a) describes AST structure of CQI code snippet "double b;" . If we delete CQI code snippet, GumTree will generate code change actions sequence as shown in Figure 6 (b). Actually the sequence of four DEL actions can be merged into only one action (DEL VariableDeclarationStatement) and we can also know the meaning of this code change in CQI code snippet.

Algorithm 2 illustrates how the algorithm extract the merged DEL actions in the actions list at a given start position. Algorithm traverses the code change action list, simply merges the related DEL actions which belong to the same AST subtree and finally return the DEL with the farthest parent node of the subtree. Besides, irrelevant DEL actions will be returned in the $deletion\_list$ if there are.

Extracting Fix Pattern

The code change actions around the CQI fix hunk collected in Section III.B are regarded as the CQI Fix Actions. However, there are always quite a lot noisy actions and it's hard to filter out all of them. Simply, we collect the code change actions that line number of AST node action operate satisfy:

$$LineNumber_{start} \leq LineNumber \leq LineNumber_{end} \quad (1)$$

The $LineNumber_{start}$ is the start line number of fix hunk, and the $LineNumber_{end}$ is the end line number of fix hunk in pre-fix code. Figure 7 show the AST-level code change actions of Fix Instance in Figure 5. This Fix Instance replace InfixExpression "$top.current.key! = key$" with InfixExpression "$Double.compare(top.current.key, key)! = 0$" .

After collecting CQI Fix Actions, we define the FP as:

$$FP = (TYPE, CCTX, FA) \quad (2)$$

where TYPE is the CQI type fixed by FP and we record the rule id as TYPE. CCTX is CQI Code Context information include CQI issue code and the tokenized textual vector of AST Nodes in CQI issue lines. FA is Fix Actions that record the AST-level operation of fix hunk. We also abstract the noisy information in FA, and mine the common Fix Patterns for all the CQI type we mined which will be introduced in Setction IV.

D. FIX PATTERN MATCHING AND FIX PATCH RECOMMENDING

First, we get the TYPE and CCTX of CQI Source Code. Then with TYPE and CCTX, we introduce a two-phase matching process to get a Fix Pattern sequence. Finally we generate fix patch by changing the source code with FA in each Fix Pattern.

The CQI source code will be scanned by SonarQube to detect the CQI instance firstly. And we parse the CQI source code to get CCTX of issue code as described in Section III.C. After that, in the first phase of matching, we filter out a Fix Pattern list in which Fix Pattern's TYPE is the same as the issue code.

Then at the second phase, we compute similarities between CCTX of CQI code and CCTX of Fix Patterns. The cosine similarity is widely adopted to evaluate the similarity between two numerical vectors. Thus, the textual vector of Fix Patterns are embedded and converted in numeric vectors with the same size by Word2Vec [35], which is a two-layer neural network to embed word. The complete similarity of 2 CCTXs will be 1 as they are expressed 0-degree angle in the vector space, while the worst similarity is 0 as 90-degree angle. For example, the cosine similarity between Modifier "$private$" and Modifier "$public$" is 0.8401263, which is the highest similarity and the second highest is Modifier "$protected$". Finally we get a Fix Pattern sequence ranked by the similarity

Algorithm 2: Algorithm of Extracting Merged DEL action

```
Input: action_list,start
Output: deletion_list
deletion_list ← null
cur ← start
cur_action ← action_list.get(cur)
deletion_list.add(cur_action) if
cur_actionislastoneofaction_list then
    returndeletion_list
end
if
cur_action.nodeisleafnodeAnd(action_list.get(cur+
1) is not DEL) then
    returndeletion_list
end
cur ← cur + 1 while cur < action_list.size() do
    next ← action_list.get(cur) if
    next.childdrencotainscur_action.node then
        temp ← deletion_list.next() while
        deletion_list.next()isnotnull do
            if next.childdrencotaintemp.node then
                deletion_list.remove(temp)
            end
            temp ← deletion_list.next()
        end
    end
    deletion_list.add(next) cur_action ← next
    cur ← cur + 1
end
returndeletion_list
```

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2019.2934975, IEEE Access

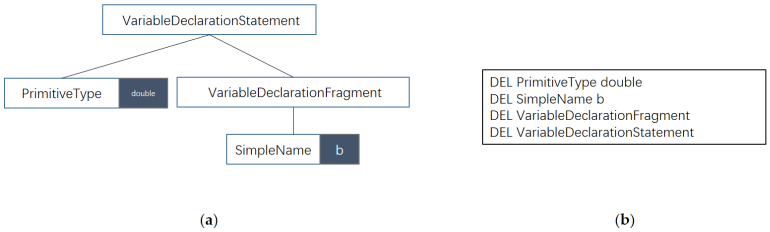


FIGURE 6. The Fix Instance collected from commit $16431a$ of $elasticsearch$ project

```
INS_MethodInvocation
INS_NumberLiteral_0
INS_SimpleName_Double
INS_SimpleName_compare
INS_QualifiedName_top.current.key
INS_SimpleName_key
DEL_QualifiedName_top.current.key
DEL_SimpleName_key
```

FIGURE 7. The Code change actions of Figure 5

TABLE 1. Statistic of OS projects used in our study

Project	Star	Loc	Commits	CQI Fix Commits
Java-design-patterns	47,810	24,630	2,179	264
RxJava	39,082	274,068	5,533	1,199
elasticsearch	41,267	111,261	46,093	8,963
Sonarqube	3,629	105,219	28,318	6,267
Search-guard	2,085	15,705	1,215	305

of CCTX and generate fix patch by changing the source code with FAs in each Fix Pattern.

IV. RESULTS AND EVALUATION

In this section, we first introduce the experiment setting in Section IV.A and show the statistics of Fix pattern and common Fix Pattern in SectionIV.B. Then we evaluate Fix Patterns with the CQI instances in Section IV.C and a real-word live study in GitHub.

A. EXPERIMENTAL SETTINGS

In this paper, we select 206 popular OSS repositories from GitHub, which is the most popular code hosting site, and SonarCloud, an online SonarQube cloud platform. We extracted 54,611 CQI Fix commits, and mined 25,424 Fix Instance. Table 1 lists statistics of projects. Star is corresponding with the GitHub users who are interested in this project, Loc describe the line of Java Code in the project and CQI Fix $Commits$ are commit that is related with CQI Fix.

B. STATISTICS

For each project, we execute the SonarQube on every CQI fix commit version and the previous version of CQI Fix. As described in Section III.A, we execute SonarQube on the pre-fix version to collect the CQI Instance. Then we set the repository to the fix version and execute SonarQube again to track the CQI status. We have collected more than 5,047,678 CQI Instances of 220 different CQI types. Figure 8 show the proportion of each CQI types, where the value of x-axis represent random id numbers of CQI types and the y-axis represents the proportion of each CQI types. Top1 CQI Type of the proportion accounts for 45.29% of all the CQI Instances, which is a coding convention issue that Public types, methods and fields (API) should be documented with Javadoc. We also find that the first 27 CQI types account for more than 90% of all CQIs. These CQI types only account for 7.12% of all SonaQube CQI types for Java. It confirm that there exist vast CQIs in Open Source Community and most CQIs are recurrent while most of them are fixed by developers.

We collected 31,013 CQI Fix Instances of 175 CQI types which ignore the false positive of fix, and the fix without code changing. Figure 9 shows the quantity distribution of fixed CQIs, where the value of x-axis represent random id numbers of CQI types which is consistent with CQI id in Figure 8 and the y-axis represents the proportion of each CQI types. We noted that fixed CQI types account for 79.54% of all detected 220 CQI types. It corroborates that despite false positive of SonarQube, developers attach importance to code quality and most CQI types are fixed manually. Thus we select this Fix Instance to mine CQI Fix Pattern following the method in Section III.C and 10 example CQI Fix Patterns is shown in Table 2. Note that the CQI type is too long and we simply reference CQI type only with its ID named by SonarQube in the following.

1) Concrete Fix Pattern

We mine one unique Fix Pattern for each CQI type in common situation and it can be used in common context unconditionally. For example, CQI 524 describes that having two branches in the same if structure with the same implementation is at best duplicate code, and at worst a coding error. As a result, a probable Fix Pattern is to combine conditions with "||" and remove the redundant block. CQI 431 describes that Java Language Specification recommends listing modifiers in the following

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2019.2934975, IEEE Access

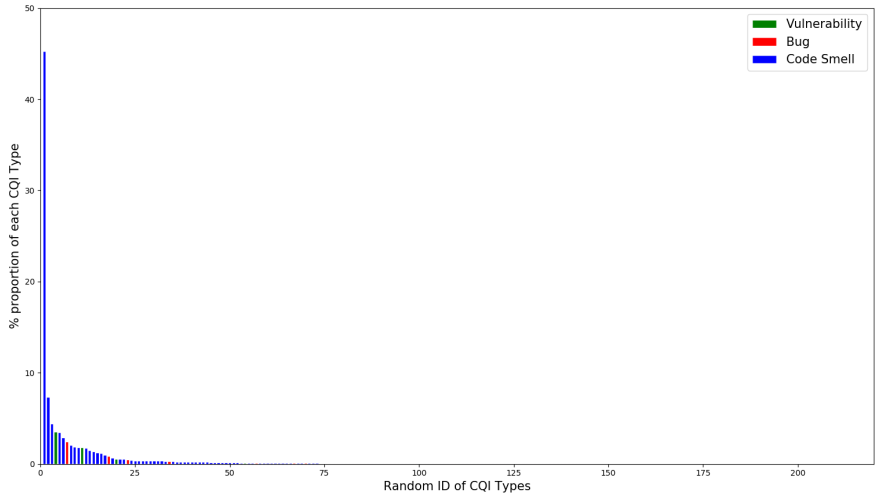


FIGURE 8. Quantity distributions of all collected CQI Instances ordered by proportion.

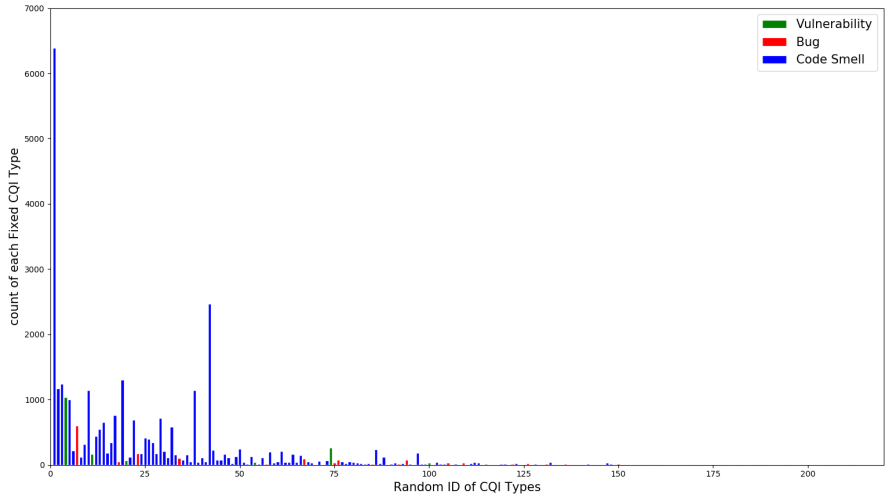


FIGURE 9. Quantity distributions of all collected fixed CQI Instances ordered by count.

order: $annotation, public, protected, private, abstract, static, final$, etc. It can be fixed by Fix Pattern that move the modifiers to comply the Java Language Specification.

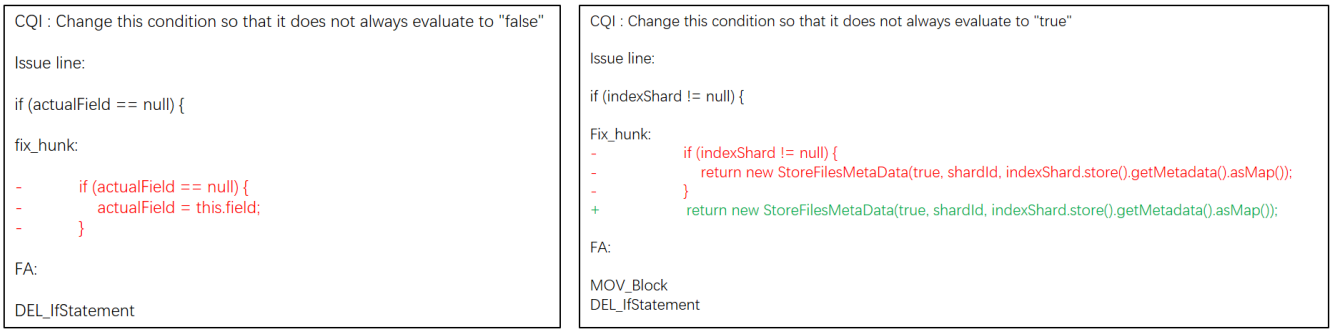
Also, there may be some different Fix Pattern for the same CQI type. CQI 419 describes that the equals method takes a generic Object as a parameter, any type of object may be passed to it, so it must check the parameter's type. We can use instanceof to check the parameter's type with if structure. Also object. getClass() can be check parameter's type.

However, some concrete Fix patterns contain some particular features and should be used in concrete context. For example, Figure 10 shows 2 different Fix Patterns of CQI 770 as is described in Table 2. It means conditional statements using a condition which cannot be anything but FALSE have the effect of making blocks of code non-functional and if the condition cannot evaluate to anything but TRUE, the conditional statement is completely redundant, and makes the code less readable. As a result, Figure 10 (a) just remove the Ifstatement to avoid dead code in the condition that is always "False", while Figure 10 (b) only remove the condition of Ifstatement and keep the block of Ifstatement.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2019.2934975, IEEE Access

TABLE 2. Examples of Common Fix Pattern

ID	CQI Type	Fix Pattern(s)
419	"equals(Object obj)" should test argument type	(1) add instanceof to check argument type (2) add getClass() to check argument type
431	Modifiers should be declared in the correct order	Move modifier(s)
480	The diamond operator ("<>") should be used	Delete type in "<>" in assignment Statement
524	Two branches in the same conditional structure should not have exactly the same implementation	Combine conditions with "||" and remove the redundant block
597	Null pointers should not be dereferenced	(1) add Ifstatement to check if it is null (2) add ternary conditional operator to check if it is null
665	Dead stores should be removed	Delete the dead code
691	Floating point numbers should not be tested for equality	Replace the relational operator with Double.compare()
697	Boolean literals should not be redundant	Delete the comparison with Boolean value
770	Conditions should not unconditionally evaluate to "TRUE"	(1) Remove Ifstatement or to "FALSE" if condition is always 'False' (2) Remove Ifstatement and keep the block if the condition is always 'True'
772	Strings literals should be placed on the left side when checking for equality	(1) remove string literals to the left side (2) replace equals with Boolean.parseBooleanExact()



(a) (b)

FIGURE 10. Concrete Fix Patterns for CQI 770. (a) is taken from $commitde1e4e$ in project elasticsearch while (b) from $commita4f974$ in project elasticsearch.

2) Non-Pattern

In this paper, we also find some CQI Fix Instances can't be extracted and mined into Fix Pattern in a syntactic way. For example, CQI 602 describes that Type parameter names should comply with a naming convention. That is, shared naming conventions make it possible for a team to collaborate efficiently. Following the established convention of single-letter type parameter names helps users and maintainers of your code quickly see the difference between a type parameter and a poorly named class. However, it's hard to rename the parameter name complying the project coding convention and infer the semantic information for it. So we think CQI 602 is Non-Pattern in syntactic way of our approach.

C. EVALUATION

When fixing CQI Instance, our approach will parse the CQI source code to get the CQI code context and then match CQI Fix Pattern to generate fix patch. To evaluate the correctness of our approach, we apply our Fix Patterns to fix the CQI Instances which aren't fixed collected by our approach in Section IV.D. Also we investigate the usefulness of our approach in Open Source Community by doing live study.

We collect a test set of unfixed CQI Instance of Top 20 CQI types which can be fixed by Fix Pattern. For each CQI types, we collect 5 unfixed instances randomly and parse the source code to get the CQI Code Context. After that, we use matching algorithm to match the Top 5 most similar Fix Pattern to generate fix patches. If there are not enough unfixed CQI Instance or matched Fix Pattern, then we collect as many as possible. Finally we execute the SonarQube to

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2019.2934975, IEEE Access

TABLE 3. the Results of Fixing Unsolved CQI Instances

ID	CQI Type	Top 1	CR@1	Top 5	CR@5
706	Useless imports should be removed	5/5	100.00%	25/25	100.00%
480	The diamond operator ("<>") should be used	5/5	100.00%	24/25	96.00%
431	Modifiers should be declared in the correct order	5/5	100.00%	17/25	68.00%
512	Sections of code should not be "commented out"	5/5	100.00%	20/25	80.00%
552	Multiple variables should not be declared on the same line	1/1	100.00%	3/5	60.00%
430	"public static" fields should be constant	1/1	100.00%	1/5	20.00%
772	Strings literals should be placed on the left side when checking for equality	5/5	100.00%	21/25	84.0%
580	Methods and field names should not be the same or differ only by capitalization	5/5	100.00%	20/25	80.00%
698	Local Variables should not be declared and then immediately returned or thrown	4/4	100.00%	11/20	55.00%
603	Standard outputs should not be used directly to log anything	3/3	100.00%	15/15	100.00%
581	Anonymous inner classes containing only one method should become lambdas	4/5	80.00%	13/25	52.00%
703	Useless parentheses around expressions should be removed to prevent any misunderstanding	4/5	80.00%	18/25	72.00%
486	Class variable fields should not have public accessibility	4/5	80.00%	19/25	76.00%
570	"@Override" annotation should be used on any method overriding (since Java 5) or implementing (since Java 6) another one	4/5	80.00%	14/25	56.00%
665	Dead stores should be removed	3/5	60.00%	14/25	56.00%
657	Unused method parameters should be removed	3/5	60.00%	15/25	60.00%
704	Unused "private" methods should be removed	3/5	60.00%	19/25	76.00%
680	Unused local variables should be removed	3/5	60.00%	17/25	68.00%
641	Collapsible "if" statements should be merged	1/3	33.33%	8/15	53.33%
602	Local variables should not shadow class fields	0/3	0.00%	0/15	0.00%
	total	68/85	80%	294/425	69.17%

validate if the Fix Pattern can fix the CQI Instance.

1) Correctness of Fix Pattern

Table 3 show the results of fixing unsolved CQI Instance. The left side in X/X of Top 1 is the fixed number while the right is number of CQI Instance. Since almost CQI Instances can be fixed in Top 5 Fix Pattern, we evaluate the correctness of all Top 5 Fix Pattern. We define CR@1 as Correctness Rate of most similar Fix Pattern, while CR@5 as Correctness Rate in Top 5 Fix Pattern. Among the Evaluated CQI Instances, 80% can be fixed by Top 1 Fix pattern and 69.17% of Top 5 Fix Patterns can be applied to fix CQI Instance. We also learn some insights from the experiment:

1) Some CQI type can almost be fixed by the matched Fix Patterns. For example, for CQI 706, which means that useless imports should be removed, the matched Fix Patterns simply delete the lines of $ImportDeclaration$. For CQI 480, the code "$List < String > strings = newArrayList < String > ()$;" can be simplified to "$List < String > strings = newArrayList <> ()$", since Java 7 introduced the diamond operator (<>) to deduce the verbosity of generic code. As a result, for the simple DEL operations, the matched Fix Pattern can perform very well.
2) Some Fix Patterns can't be applied to fix the CQI Instances even with high similarity of context in CQI line. For CQI 570, most matched Fix Pattern is that Insert "@Override" Annotation. However, applying fix pattern that simply removes the method will fail if there exists method invocation in project.

TABLE 4. partial OSS projects of live study

Project	Star	Loc	Commits
swarm-plugin	100	1,583	1,215
mybatis-dynamic-sql	304	14,066	514
gapic-generator	101	53,893	2,295
pentaho-kettle	2,714	793,746	21,166
lunar-unity-console	397	17,971	894
DSpace	447	187,987	10,744

2) Live Study

We investigate the usefulness of our approach in OSS Community. Table 4 lists partial OSS projects in live study which cover small, median and large scale. We fork the projects to our branch and execute SonarQube on the projects to detect CQI Instances. Then we randomly select the CQI types which can be fixed by Fix Pattern of our approach. Then we generate the fix patches and scan them again to ensure the CQI has been fixed. Finally, we create pull requests to projects' master branch. We describe the CQI details and Fix Pattern in the pull requests.

Table 5 shows that 17 pull request are pushed to projects. In these 17 pull requests, 5 of them are ignored and requiring reviews, 1 of them is rejected, 1 is approved and wait for merging and 10 of them are merged by developers.

In this live study, we also find that: We used CQI Density (CQID), which means the CQI numbers per Loc(Line of Code), t to measure the code quality of project. Results show that CQID of largest project gapic-generator is 10.25%, the CQID of median scale $mybatis-dynamic-sql$ is 11.38%, and the CQID of the smallest scale project, swarm-plugin is

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2019.2934975, IEEE Access

TABLE 5. Fixed CQI Types in our live study

ID	CQI Type
655	Public constants and fields initialized at declaration should be "static final" rather than merely "final"
118	Empty statements should be removed
419	"equals(Object obj)" should test argument type
431	Modifiers should be declared in the correct order
581	Anonymous inner classes containing only one method should become lambdas
597	Null pointers should not be dereferenced
698	Local Variables should not be declared and then immediately returned or thrown
770	Conditions should not unconditionally evaluate to "TRUE" or to "FALSE"

TABLE 6. Results of live study

Project	Pull Request	Waited	Rejected	Approved	Merged
swarm-plugin	3	0	0	0	3
mybatis-dynamic-sql	3	0	0	0	3
gapic-generator	3	0	0	0	3
pentaho-kettle	1	1	0	0	0
lunar-unity-console	1	0	1	0	0
DSpace	1	0	0	1	0
rrd4j	1	1	0	0	0
OpenClinica	1	1	0	0	0
asciidoctorj	1	0	0	0	1
wenku	1	0	0	0	1
git-merge-repos	1	1	0	0	0
total	17	4	1	1	11

12.12%. It notes that large scale projects are well-maintained and exist less CQI.

Most of the CQI Fix are accepted by the developers unconditionally. For example, we create Pull Request 2769 in $gapic - generator$ project, which generates a fix patch for CQI 770 described in Table 6. In method getFormattedPrintArgName, the parameter type has been referenced before if structure. If $type$ is null, $NullPointException$ would be thrown. As a result, the condition "$type == null$" will always be "false" here. So the fix path is that remove condition "$type == null$" and logical operator "||". After creating this Pull Request, the developer review it with a comment "$LGTM$", which means it Looks Good To Me, and merge our Pull Request as shown in Figure 11.

Some CQI Instances have better resolution in particular code context. We create Pull Request 85 in swarm-plugin to fix CQI 655 described in Table 6. This CQI means that making a public constant just final as opposed to static final leads to duplicating its value for every instance of the class, uselessly increasing the amount of memory required to execute the application. We insert final modifier in fix patch and create the Pull Request. However, developer reviewed the PR and comment that there may be a better resolution to fix this CQI, as shown in Figure 12. As a result, we generate the modified fix patch in Figure 12 which directly removed the variable $labelFileWatcherThread$ and the PR is approved and merged by developers as shown in Figure 13.

V. DISCUSSION

There are two potential threats to the validity of this work: internal and external validity.

A. INTERNAL VALIDITY

The definition of CQI is limited. CQIs consist of 3 parts: Bugs, Vulnerabilities and Code smell [37] in descending order of severity. The validity of our approach is built on the tool we used - SonarQube. The static model are based on the SQALE methodology, which is a public methodology to support the evaluation of a software applications source code in the most objective, accurate, reproducible and automated possible way [36]. One of the potential issue is the false positive and we find some cases in the evaluation. For instance, when analyzing Android project, SonarQube reported the naming convention issue for the R file, which is an auto-generated resource file of Android Development Toolkit, saying that the naming style of underscore (e.g., m_image) is inconsistent with Java convention (e.g. $mImage$). As a result, SonarQube reported code quality issues on convention at R file, which are also involved in our data set.

B. EXTERNAL VALIDITY

Projects analyzed may not be representative. We analyzed 206 OSS projects in GitHub and hence might not be representative of all development context. It's possible that the different development process, different goal of development could lead to different Fix Patterns. Besides, we only analyze the projects of Java language. Other programming languages may have different Fixing Pattern.

Fixing data is incomplete. Our approach is based on the $gitlog$ command, which compares a source code file of two versions, so the Fix Patterns are only file-by-file based. The cross-file CQIs and their Fix Patterns are not revealed. For instance, SonarQube reported CQI that the default unnamed

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2019.2934975, IEEE Access

Merged Patch:

```
   public String getFormattedPrintArgName(
      ImportTypeTable typeTable, TypeModel type, String variable, List<String> accessors) {
      Preconditions.checkArgument(
         !type.isRepeated() && !type.isMap(),
         "Expected non-repeated fields in print statement, found %s",
         type.getTypeName());
      String arg = "$" + variable + String.join("", accessors);
-     if (type == null || !(type instanceof ProtoTypeRef) || type.isPrimitive()) {
+     if (!(type instanceof ProtoTypeRef) || type.isPrimitive()) {
```

FIGURE 11. Example of merged patch in $gapic - generator$ project.

Fix Patch:

```
-   private final Thread labelFileWatcherThread = null;
+   private static final Thread labelFileWatcherThread = null;
```

Modified Fix Patch:

```
-   private final Thread labelFileWatcherThread = null;
```

FIGURE 12. Example of fix patch and modified fix patch in $swarm - plugin$ project.

FIGURE 13. Comment in Pull Request 85 of $swarm - plugin$ project

package should not be used, when finding file is not in a named package. The Fix Pattern is that move the file to a named package and it cannot be extracted by our approach. We may not fix some CQIs because of the missing Fix Patterns.

VI. CONCLUSION AND FUTURE WORK

In this paper, we propose a history-driven approach to automatically fix CQIs. To the best of our knowledge, we propose the first approach that automatically fixes CQIs using history-driven approach. On the one hand, our approach can improve the efficiency of converging massive contributions in OSS community. On the other hand, by utilizing fix knowledge (e.g. Fix Patterns) learned from code change history in OSS code repositories, our approach can supply diversified, real-world fix guide for CQI to be fixed. We collected 31,013 CQI fixing Instances detected by SonarQube from 206 OSS projects and extracted 68 common fix patterns for 56 CQI types. Our experiment shows that the average correctness of fix pattern is 80% in top 1 fix patch and 69.17% of Top 5 Fix patterns can be used to fix CQI Instance. We also conduct a live study in GitHub and the results show that developers

approved or merged 11 of 17 CQI fixing patches. In future work, we plan to mine more Fix Patterns and develop a pratical tool to automatically fix CQIs in the OSS Community in the future.

ACKNOWLEDGMENT

We acknowledge the developers for participating our live study in GitHub.

REFERENCES

[1] Gaudin O, SonarSource. Continuous inspection: A paradigm shift in software quality management. Technical Report, SonarSource S.A., Switzerland 2013.

[2] Lu Y , Mao X , Li Z , et al. Internal quality assurance for external contributions in GitHub: An empirical investigation[J]. Journal of Software: Evolution and Process, 2018, 30(4):e1918.

[3] Nagappan N, Williams L, Hudepohl J, Snipes W, Vouk M. Preliminary results on using static analysis tools for software inspection. the 15th IEEE International Symposium on Reliability Engineering 2004; :429âĂŞ439

[4] Tonella P, Abebe SL. Code quality from the programmerâĂŹs perspective. PoS 2008; 001

[5] Lu Y, Mao X, Li Z, et al. Does the Role Matter? An Investigation of the Code Quality of Casual Contributors in GitHub[C]// Software Engineering Conference. IEEE, 2017:49-56.

[6] [Online].Available: https://help.eclipse.org/2019-06/index.jsp

[7] Yu Y, Wang H, Yin G, et al. Reviewer recommendation for pull-requests in GitHub: What can we learn from code review and bug assignment?[J]. Information Software Technology, 2016. 74(C):204-218.

[8] [Online].The State of the Octoverse 2018, https://octoverse.github.com, 2018.

[9] Liu K , Kim D , BissyandÃl', TegawendÃl' F, et al. Mining Fix Patterns for FindBugs Violations[J]. IEEE Transactions on Software Engineering, 2017.

[10] X.-B. D. Le, D. Lo, and C. Le Goues, "History driven program repair," in Proceedings of the International Conference on Software Analysis, Evolution, and Reengineering (SANER), 2016, pp. 213âĂŞ224, doi: 10.1109/SANER.2016.76.

[11] X.-B. D. Le, "Towards efficient and effective automatic program repair," in IEEE/ACM International Conference on Automated Software Engineering (ASE), 2016, pp. 876âĂŞ879, doi: 10.1145/2970276.2975934.

[12] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest, Automatically finding patches using genetic programming, in Proceedings of the 31st International Conference on Software Engineering, ser. ICSE 09. IEEE Computer Society, 2009, pp. 364374.

[13] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, GenProg: A Generic Method for Automatic Software Repair, Software Engineering, IEEE Transactions on, vol. 38, no. 1, pp. 5472, Feb. 2012.

[14] Gazzola L, Micucci D, Mariani L. Automatic software repair: A survey[J]. IEEE Transactions on Software Engineering, 2017.

[15] Y. Wei, Y. Pei, C. A. Furia, L. S. Silva, S. Buchholz, B. Meyer, and A. Zeller, Automated fixing of programs with contracts, in Proceedings of the 19th international symposium on Software testing and analysis. Trento, Italy: ACM, 2010, pp. 6172.

[16] J. Xuan, M. Martinez, F. DeMarco, M. Clment, S. L. Marcote, T. Durieux, D. L. Berre, and M. Monperrus, Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs, IEEE Transactions on Software Engineering, vol. 43, no. 1, pp. 3455, Jan. 2017.

[17] V. Dallmeier, A. Zeller, and B. Meyer, Generating Fixes from Object Behavior Anomalies, in 24th IEEE/ACM International Conference on Automated Software Engineering, 2009. ASE 09. IEEE, Nov. 2009, pp. 550554.

[18] S. Forrest, T. Nguyen, W. Weimer, and C. Le Goues, A genetic programming approach to automated software repair, in Proceedings of the Annual Conference on Genetic and Evolutionary Computation (GECCO), 2009, pp. 947954, doi: 10.1145/1569901.1570031.

[19] W. Weimer, S. Forrest, C. Le Goues, and T. Nguyen, Automatic program repair with evolutionary computation, Communications of the ACM (CACM), vol. 53, no. 5, pp. 109116, 2010, doi: 10.1145/1735223.1735249.

[20] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer, A systematic study of automated program repair: Fixing 55 out of 105 bugs for $ 8 each, in Proceedings of the International Conference on Software Engineering (ICSE), 2012, pp. 313, doi: 10.1109/ICSE.2012.6227211.

[21] C. Le Goues, Automatic program repair using genetic programming, Ph.D. dissertation, University of Virginia, 2013.

[22] R. Kou, Y. Higo, and S. Kusumoto, A capable crossover technique on automatic program repair, in Proceedings of the International Workshop on Empirical Software Engineering in Practice (IWESEP), 2016, pp. 4550, doi: 10.1109/IWESEP.2016.15.

[23] Ji T, Chen L, Mao X, et al. Automated Program Repair by Using Similar Code Containing Fix Ingredients[C]//Computer Software and Applications Conference (COMPSAC), 2016 IEEE 40th Annual. IEEE, 2016, 1: 197-202.

[24] 19. Koyuncu A, Liu K, Bissyand T F, et al. FixMiner: Mining Relevant Fix Patterns for Automated Program Repair[J]. arXiv preprint arXiv:1810.01791, 2018.

[25] Gupta R, Pal S, Kanade A, et al. DeepFix: Fixing Common C Language Errors by Deep Learning[C]//AAAI. 2017: 1345-1351.

[26] H. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, Semfix: Program repair via semantic analysis, in Proceedings of the International Conference on Software Engineering (ICSE), 2013, pp. 772 781, doi:10.1109/ICSE.2013.6606623.

[27] S. Mechtaev, J. Yi, and A. Roychoudhury, Directfix: Looking for simple program repairs, in Proceedings of the International Conference on Software Engineering (ICSE), 2015, pp. 448458, doi: 10.1109/ICSE.2015.63.

[28] Kim S, Pan K, Whitehead E E J. Memories of bug fixes[C]// ACM Sigsoft International Symposium on Foundations of Software Engineering. ACM, 2006:35-45.

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

[30] E. Zangerle, W. Gassler, and G. Specht. Using tag recommendations to homogenize folksonomies in microblogging environments. In SocInfo11, pages 113126, 2011.

[31] D. Kim, J. Nam, J. Song, and S. Kim, Automatic Patch Generation Learned from Human-written Patches, in Proceedings of the 2013 International Conference on Software Engineering. Piscataway, NJ, USA: IEEE Press, 2013, pp. 802811.

[32] H. Zhong and Z. Su, An Empirical Study on Real Bug Fixes, in Proceedings of the 37th International Conference on Software Engineering - Volume 1. Piscataway, NJ, USA: IEEE Press, 2015, pp. 913923.

[33] P. Avgustinov, A. I. Baars, A. S. Henriksen, and G. Lavender, Tracking static analysis violations over time to capture developer characteristics, in Proc. the IEEE Intl Conf. on Software Engineering, 2015, pp. 437447.

[34] Falleri J R, Morandat F, Blanc X, et al. Fine-grained and accurate source code differencing[C]//Proceedings of the 29th ACM/IEEE international conference on Automated software engineering. ACM, 2014: 313-324.

[35] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," CoRR, vol. abs/1301.3781, 2013.

[36] J. L. Letouzey, The sqale method definition document, in 3rd Intl Workshop on Managing Technical Debt (MTD), 2012, pp. 31 36.

[37] Tufano, M.; Palomba, F.; Bavota, G.; Oliveto, R.; Di Penta, M.; De Lucia, A.; Poshyvanyk, D. (2015-05-01). "When and Why Your Code Starts to Smell Bad" . 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE). 1: 403414.