Learning Quick Fixes from Code Repositories

Reudismam Rolim
UFERSA
Pau dos Ferros, RN, Brazil
reudismam.sousa@ufersa.edu.br

Gustavo Soares
Microsoft
Redmond, WA, USA
gsoares@microsoft.com

Rohit Gheyi
UFCG
Campina Grande, PB, Brazil
rohit@dsc.ufcg.edu.br

Titus Barik
Microsoft
Redmond, WA, USA
titus.barik@microsoft.com

Loris D'Antoni
University of Wisconsin
Madison, WI, USA
loris@cs.wisc.edu

ABSTRACT

Code analyzers such as Error Prone and FindBugs detect code patterns symptomatic of bugs, performance issues, or bad style. These tools express patterns as quick fixes that detect and rewrite unwanted code. However, it is difficult to come up with new quick fixes and decide which ones are useful and frequently appear in real code. We propose to rely on the collective wisdom of programmers and learn quick fixes from revision histories in software repositories. We present Revisar, a tool for discovering common Java edit patterns in code repositories. Given code repositories and their revision histories, Revisar (*i*) identifies code edits from revisions and (*ii*) clusters edits into sets that can be described using an edit pattern. The designers of code analyzers can then inspect the patterns and add the corresponding quick fixes to their tools. We ran Revisar on nine popular GitHub projects, and it discovered 89 useful edit patterns that appeared in 3 or more projects. Moreover, 64% of the discovered patterns did not appear in existing tools. We then conducted a survey with 164 programmers from 124 projects and found that programmers significantly preferred eight out of the nine of the discovered patterns. Finally, we submitted 16 pull requests applying our patterns to 9 projects and, at the time of the writing, programmers accepted 7 (63.6%) of them. The results of this work aid toolsmiths in discovering quick fixes and making informed decisions about which quick fixes to prioritize based on patterns programmers actually apply in practice.

KEYWORDS

Software Engineering, Education, Survey.

ACM Reference Format:
Reudismam Rolim, Gustavo Soares, Rohit Gheyi, Titus Barik, and Loris D'Antoni. 2021. Learning Quick Fixes from Code Repositories. In *Brazilian Symposium on Software Engineering (SBES '21), September 27-October 1, 2021, Joinville, Brazil.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3474624.3474650

1 INTRODUCTION

Programmers often detect code patterns that may lead to undesired behaviors (e.g., inefficiencies) and apply simple code edits to "fix" these patterns. These patterns are often hard to spot because they depend on the style and properties of the programming language in use. Tools such as Error Prone, FindBugs, and PMD help programmers by automatically detecting and sometimes removing several suspicious code patterns, potential bugs, or instances of bad code style. For example, PMD can detect instances where the method `size` is used to check whether a list is empty and proposes to replace it with the method `isEmpty`. For the majority of collections, these two ways to check emptiness are equivalent, but for some collections—e.g., `ConcurrentSkipListSet`—computing the size of a list is not a constant-time operation [37]. We refer to these kinds of edit patterns as *quick fixes*.

All the aforementioned tools rely on a predefined catalog of quick fixes (usually expressed as rules), each used to detect and potentially fix a pattern. These catalogs have to be updated often due to new language features (e.g., new constructs in new Java versions), new style guidelines, or simply due to the discovery of new patterns. However, coming up with what edit patterns are useful and common is a challenging and time-consuming task that is currently performed in an ad-hoc fashion—i.e., new rules for quick fixes are added on a as-needed basis.

> The lack of a systematic way of discovering new quick fixes makes it hard for code analyzers to stay up-to-date with the latest code practices and language features.

For example, consider the edit pattern applied to the Apache Ant code in Figure 1a. The original code contains three expressions of the form `x.equals("str")` that compare a variable `x` of type string to a string literal `"str"`. Since the variable `x` may contain a `null` value, evaluating this expression may cause a `NullPointerException`. In this particular revision, a programmer from this project addresses the issue by exchanging the order of the arguments of the `equals` method—i.e., by calling the method on the string literal. This edit fixes the issue since the `equals` method checks whether the parameter is `null`. This edit is common and we discovered it occurs in three industrial open source projects across GitHub repositories: Apache Ant, Apache Hive, and Google ExoPlayer. Since the pattern appears in such large repositories, it could also be useful to other programmers who may not know

```
   //...
-  } else if (args[i].equals("--launchdiag")) {
+  } else if ("--launchdiag".equals(args[i])) {
     launchDiag = true;
-  } else if (args[i].equals("--noclasspath")
-    || args[i].equals("-noclasspath")) {
+  } else if ("--noclasspath".equals(args[i])
+    || "-noclasspath".equals(args[i])) {
   //...
```

(a) Concrete edits applied to the Apache Ant source code in the project commit history (https://github.com/apache/ant/commit/b7d1e9b).

```
@BeforeTemplate                   @AfterTemplate
boolean b(String v1,              boolean a(String v1,
 StringLiteral v2) {        →      StringLiteral v2) {
  return v1.equals(v2);             return v2.equals(v1);
}                                 }
```

(b) Abstract quick fix in Refaster-like syntax for edits in Figure 1a.

Figure 1: Revisar (a) mines concrete edits from the code repository history in a project and (b) discovers abstract quick fixes from these edits.

about it. Despite its usefulness, a quick fix rule for this edit pattern is not included in the catalog of largely used code analyzers, such as FindBugs, and PMD. Remarkably, even though the edit is applied in Google repositories, this pattern is absent in Error Prone, a Google code analyzer that is internally used on the Google's code base.

Key insight Our key insight is that we can "discover" useful patterns and quick fixes by observing how programmers modify code in real repositories with large user bases. In particular, we postulate an edit pattern performed by many programmers across many projects is likely to reveal a good quick fix.

Our technique In this work, we propose Revisar, a technique for automatically discovering common Java code edit patterns in code repositories. Given code repositories as input, Revisar identifies simple edit patterns by comparing consecutive revisions in the revision histories. The most common edit patterns—i.e., those performed across multiple projects—can then be inspected to detect useful ones and add the corresponding quick fixes to code analyzers. For example, Revisar was able to *automatically* analyze the concrete edits in Figure 1 and generate the quick fix in Figure 1b. We also sent Pull Requests (PRs) applying this quick fix to other parts of the code in the Apache Ant and Google ExoPlayer projects, and these PRs were accepted.

Contributions This paper makes the following contributions:

- Revisar, an automatic technique for discovering quick fixes in large repositories. Revisar uses concepts such as Abstract Syntax Trees (AST) edits and *d*-caps, and it applies the technique of anti-unification from inductive logic programming to mine edit patterns (§ 2).
- A mixed-methods evaluation of the Revisar effectiveness at discovering quick fixes and the quality of the discovered quick fixes (§ 3 and 4). When ran on nine popular GitHub projects, Revisar discovered 89 edit patterns appearing in 3 or more projects. Moreover, 64% of the discovered patterns is absent in existing tools. Through a survey on a subset of the discovered patterns, we showed programmers significantly preferred 8/9 (89%) of our patterns. Finally, programmers accepted 7/11 (63.6%) PRs applying our patterns to their projects, showing they are willing to apply our patterns.

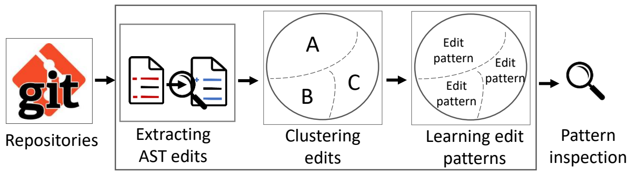


Figure 2: Revisar's work-flow.

2 REVISAR

We now describe Revisar, our technique for learning quick fixes from code repositories. Given repositories as input, Revisar: (*i*) identifies concrete code edits by comparing pairs of consecutive revisions (§ 2.1), (*ii*) clusters edits into sets that can be described using the same edit pattern and learns an edit pattern for each cluster (§ 2.2 and 2.3). Figure 2 shows the work-flow of Revisar.

2.1 Extracting concrete AST edits

The initial input of Revisar is a set $revs = \{R_1, \ldots, R_n\}$ where each R_i is a revision history $r_1 r_2 \cdots r_k$ from a different project (i.e., a sequence of revisions). For each pair (r_i, r_{i+1}) of consecutive revisions, Revisar analyzes the differences between the ASTs[1] of r_i and r_{i+1} and uses a Tree Edit Distance (TED) algorithm to identify a set of tree edits $\{e_1, e_2, ..., e_l\}$ that when applied to the AST t_i corresponding to r_i yields the AST t_{i+1} corresponding to r_{i+1}. In our setting, an edit is one of the following:

insert(x, p, k): insert a leaf node x as k^{th} child of parent node p. The current children at positions $\geq k$ are shifted one position to the right.

delete(x, p, k): delete a leaf node x which is the k^{th} child of parent node p. The deletion may cause new nodes to become leaves when all their children are deleted. Therefore we can delete a whole tree through repeated bottom-up deletions.

update(x, w): replace a leaf node x by a leaf node w.

move(x, p, k): move tree x to be the k^{th} child of parent node p. The current children at positions $\geq k$ are shifted one position to the right.

Given a tree t, let $s(t)$ be the set of nodes in the tree t. Intuitively, solving the TED problem amounts to identifying a partial mapping $M : s(t) \mapsto s(t')$ between source tree t and target tree t' nodes. The mapping can then be used to detect which nodes are preserved by the edit and in which positions they appear. When a node $n \in s(t)$ is not mapped to any $n' \in s(t')$, then n was deleted.

Of the many existing tools available for computing tree edits over Java code, Revisar builds on GumTree [12], a tool that focuses on finding edits representative of those intended by programmers instead of just finding the smallest possible set of tree

[1]Revisar uses Eclipse JDT to extract partial type annotations of the ASTs when it is possible to extract them. In our implementation, we use these type annotations to create a richer AST with type information—i.e., every node has a child describing its type. For simplicity, we omit this detail.

edits. To give an example of edits computed by GumTree, let's look at the first two lines in Figure 1. Figure 3 illustrates the ASTs corresponding to `args[i].equals("--launchdiag")` and `"--launchdiag".equals(args[i])`, respectively.

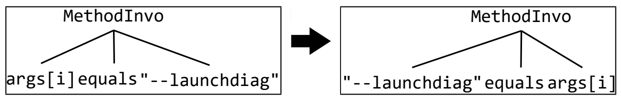


Figure 3: Before-after version for the first edited line of code.

To produce the modified version of the code in line 2 at Figure 1a, GumTree learned four edits:

```
insert("--launchdiag", MethodInvo, 0)
insert(equals, MethodInvo, 1)
delete(equals, MethodInvo, 3)
delete("--launchdiag", MethodInvo, 3)
```

These edits move the string literal `"--launchdiag"` and the `equals` node so that they appear in front of `args[i]`.

For our purposes, GumTree's edits are at too low granularity since they modify nodes instead of expressions. In particular, we want to detect edits to entire subtrees—e.g., an edit to a method invocation `args[i].equals("--launchdiag")` instead of to individual nodes inside it. Revisar identifies subtree-level edits by grouping edits belonging to the same connected components. Concretely, Revisar identifies connected components by analyzing the parent-child and sibling relationships between the nodes appearing in the tree edits. Two edits e_1 and e_2 share the same component if (*i*) the two nodes x_1 and x_2 modified by e_1 and e_2 have the same parent, or (*ii*) the x_1 (resp. x_2) is the parent of x_2 (resp. x_1). For instance, the previously shown edits are associated to two nodes x_1 = `"--launchdiag"` and x_2 = `equals`. These nodes are connected since they have the same parent node—i.e., the method invocation.

Once the connected components are identified, Revisar can use them to identify tree-to-tree mappings between subtrees inside the original and modified trees like the one showed in Figure 3. We call this mapping a *concrete edit*. A concrete edit is a pair (i, o) consisting of two components (*i*) the tree i in the original version of the program, and (*ii*) the tree o in the modified version of the program. This last step completes the first phase of our algorithm, which, given a set of revisions $\{R_1, \ldots, R_n\}$, outputs a set of concrete edits $\{(i_1, o_1), \ldots, (i_k, o_k)\}$.

2.2 Clustering concrete edits

We now show how Revisar groups concrete edits into clusters sharing the same edit pattern. Revisar's clustering algorithm receives a set of concrete edits $\{(i_1, o_1) \ldots (i_n, o_n)\}$ and uses a greedy approach. The clustering algorithm starts with an empty set of clusters. Then, for each concrete edit (i_k, o_k) and for each cluster c, Revisar checks, using the algorithm from Section 2.3, if adding (i_k, o_k) to the concrete edits of cluster c gives an edit pattern. When this happens, the cluster c is added to a set of cluster candidates and the cost of adding (i_k, o_k) to c is computed. Revisar then adds (i_k, o_k) to candidate cluster of minimum cost, or it creates a new cluster with just (i_k, o_k) if no candidate exists. The complexity of this algorithm is $O(n^2)$ where n is the number of edits since, for each edit, we have to search which cluster the edit should be included in. Instead of comparing all permutations of pairs of concrete edits, which is prohibitive, we compare an edit against each cluster. Although we lose precision, we improve performance.

When multiple clusters can receive a new concrete edit, we use a cost function to choose the appropriate cluster. The cost of adding an edit (i_k, o_k) to cluster c with corresponding template τ is computed as follows. First, Revisar anti-unifies τ and the tree in the concrete edit we are trying to cluster. Let α_k be the substitution for the result of the anti-unification and let $\alpha_k(?_i)$ be the tree substituting hole $?_i$. We define the size of a tree as the number of leaf nodes inside it, which intuitively captures the number of names and constants in the AST. We denote the cost of an anti-unification as the sum of the sizes of each $\alpha_k(?_i)$ minus the total number of holes (the same metric proposed by Bulychev et al. [6] in the context of clone detection). Intuitively, we want the sizes of substitutions to be small—i.e., we prefer more specific templates. For instance, assume we have a cluster consisting of a single tree `args[i].equals("--launchdiag")`. Upon receiving a new tree `args[i].equals("--noclasspath")`, anti-unifying the two trees yields the template `args[i].equals(`$?_1$`)` with substitutions α_1=\{$?_1$=`"--launchdiag"`\} and α_2=\{$?_1$=`"--noclasspath"`\}. We have concrete nodes `"--launchdiag"` and `"--noclasspath"` each of size one, and a single hole $?_1$. The cost will be $2 - 1 = 1$. The final cost of adding an edit to a cluster is the sum of the cost to anti-unify τ_i and i_k and the cost to anti-unify τ_o and o_k.

Predicting promising clusters For large repositories, the total number of concrete edits may be huge and it will be unfeasible to compare all edits to compute the clusters. To address this problem, Revisar only clusters concrete edits which are "likely" to produce an edit pattern. In particular, Revisar uses *d-cap*s [2, 11, 35], a technique for identifying repetitive edits. Given a number $d \geq 1$ and a template τ, a d-cap is a tree-like structure obtained by replacing all subtrees of depth d and left nodes in the template τ with holes. The d-cap works as a hash-index for sets of potential clusters. For instance, let's look at the left-hand side of Figure 3, which is the tree representation of the node `args[i].equals("--launchdiag")`. A 1-cap replaces all the nodes at depth one with holes. For our example, `args[i]`, `equals`, and `"--launchdiag"` will be replaced with holes, outputting the d-cap $?_1 . ?_2(?_3)$. Revisar uses the d-caps as a pre-step in the clustering algorithm. For all concrete edits with the same d-cap for the input tree i_k and for the output tree o_k, Revisar uses the clustering algorithm described in Section 2.2 to compute the clusters for all concrete edits in these d-cap. This heuristic makes our clustering algorithm practical as it avoids considering all example combinations. However, it also comes at the cost of sacrificing completeness—i.e., two concrete edits for which there is a common edit pattern might be placed in different clusters.

2.3 Learning edit patterns

Once Revisar has identified concrete edits—i.e., pairs of trees $\{(i_1, o_1) \ldots (i_n, o_n)\}$—it tries to group "similar" concrete edits to generate an edit pattern consistent with all the edits in each group. An *edit pattern* is a rule $r = \tau_i \mapsto \tau_o$ with two components: (*i*) the template τ_i, which is used to decide whether a subtree t in the code can be transformed using the rule r, (*ii*) the template τ_o, which describes how the tree matching τ_i should be transformed by r.

A template τ is an AST where leaves can also be holes (variables) and a tree t matches the template τ if there exists a way to assign concrete values to the holes in τ and obtain t—denoted $t \in L(\tau)$. Given a template τ over a set of holes H, we use α to denote a substitution from H to concrete trees and $\alpha(\tau)$ to denote the application of the substitution α to the holes in τ. Figure 4 shows the first two concrete edits from Figure 1 and the templates τ_i and τ_o describing the edit pattern obtained from these examples. Here, the template τ_i matches any expression calling the method `equals` with first argument `args[i]` and any possible second argument. The two substitutions α_1={$?_1$=`"--launchdiag"`} and α_2={$?_1$=`"--noclasspath"`} yield the expressions $\alpha_1(\tau_i)$ =`args[i].equals("--launchdiag")` and $\alpha_2(\tau_i)$ =`args[i].equals("--noclasspath")`, respectively. The template τ_o is similar to τ_i and note that the hole $?_1$ appearing in τ_o is the same as the one appearing in τ_i.

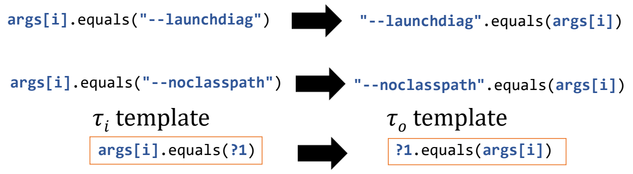

args[i].equals("--launchdiag")
"--launchdiag".equals(args[i])
args[i].equals("--noclasspath")
"--noclasspath".equals(args[i])
τ_i template
τ_o template
args[i].equals(?1)
?1.equals(args[i])

Figure 4: Concrete edits and their input-output templates.

Definition 1. *Given a set of concrete edits $S = \{(i_1, o_1), \ldots, (i_n, o_n)\}$, an edit pattern $r = \tau_i \mapsto \tau_o$, is consistent with S if: (i) the set of holes in τ_o is a subset of the set the holes appearing in τ_i (ii) for every (i_k, o_k) there exists a substitution α_k such that $\alpha_k(\tau_i) = i_k$ and $\alpha_k(\tau_o) = o_k$.*

Our goal now is to compute a template τ_i such that every AST i_j can match the template τ_i—i.e., $i_j \in L(\tau_i)$. In general, the same set of ASTs can be matched by multiple different templates, which could contain different numbers of nodes and holes. Typically, a template with more concrete nodes and fewer holes is more precise—i.e., will match fewer concrete ASTs—whereas a template with few concrete nodes will be more general—e.g., $?_1$`.equals(`$?_2$`)` is more general than `arg[i].equals(`$?_1$`)`. Among the possible templates, we want the *least general template*, which preserves the maximum common nodes for a given set of trees. The idea is to preserve the maximum amount of shared information between the concrete edits. Even when an edit is too specific, we will obtain the desired template when provided with appropriate concrete edits. In our running example, when encountering an expression of the form `x.equals("abc")`, we will obtain the desired, more general template $?_1$`.equals(`$?_2$`)`.

Remarkably, the problem we just described is tightly related to the notion of anti-unification used in logic programming [2, 3]. Given two trees t_1 and t_2, the anti-unification algorithm produces the least general template τ for which there exist substitutions α_1 and α_2 such that $\alpha_1(\tau) = t_1$ and $\alpha_2(\tau) = t_2$. In our tool we use the implementation of anti-unification from Baumgartner et al. [3], which runs in linear time. Using this algorithm, we can generate the least general templates τ_i and τ_o that are consistent with the input and output trees in the concrete edits. For now, the two templates will have distinct sets of holes, but each template can contain the same hole in multiple locations.

At this point, we have the template for the inputs τ_i and the template for the output τ_o. However, Revisar needs to analyze whether these templates describe an edit pattern $r = \tau_i \mapsto \tau_o$—i.e., whether there is a way to map the holes of τ_i to the ones of τ_o. This mapping can be computed by finding, for every hole $?_2$ in τ_o, a hole $?_1$ in τ_i that applies the same substitution with respect to all the concrete edits. To illustrate a case where finding a mapping is not possible, let's look at Figure 5. Although we can learn templates τ_i and τ_o, these templates cannot describe an edit pattern since it is impossible to come up with a mapping between the holes of the two templates that is consistent with all the substitutions. In this case, the substitution for $?_1$ in τ_i is incompatible with the substitution for $?_2$ in τ_o because `"--noclasspath"` is mapped to `"-main"`, but the content of these substituting trees differs. In addition, in our implementation, we avoid to group concrete edits that are compatible, accordingly to our definition but apply to different methods. For instance, the concrete edits (`args[i].equals("--launchdiag")`, `"--launchdiag".equals(args[i])`) and (`args[i].equalsIgnoreCase("--launchdiag")`, `"--launchdiag".equalsIgnoreCase(args[i])`) should not be in the same cluster since they are applied to the equals and `equalsIgnoreCase` methods, respectively.

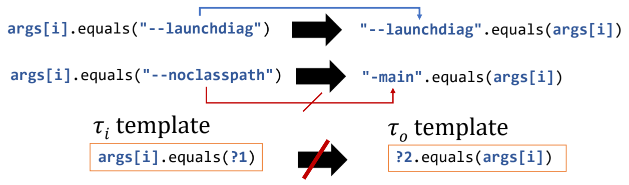


Figure 5: Incompatible concrete edits.

Since the templates are the least general, if no mapping between the holes exists, there exists no edit pattern consistent with the concrete edits—i.e., our algorithm, given a set of edits, finds a rule in our format consistent with the edits if and only if one exists. Therefore, Revisar finds all correct rules in our format and does not miss potential ones.

Theorem 1 (Soundness and Completeness). *Given a set of concrete edits $S = \{(i_1, o_1), \ldots, (i_n, o_n)\}$, Revisar returns an edit pattern $r = \tau_i \mapsto \tau_o$ consistent with S if and only if some edit pattern $r' = \tau_i' \mapsto \tau_o'$ consistent with S exists.*

3 METHODOLOGY

In this section, we describe our evaluation methodology.[2]

3.1 Research Questions

We investigate the following three research questions:

RQ1 How effective is Revisar in identifying quick fixes?
RQ2 Do developers prefer quick fixes discovered by Revisar?
RQ3 Do developers adopt quick fixes discovered by Revisar?

[2]Data available at https://sites.google.com/view/revisar/

The answer to RQ1 characterizes the edit patterns that REVISAR is able to discover, and whether these edit patterns can be framed in terms of existing code analysis tool rulesets. The answers to RQ2 and RQ3 address whether these identified edit patterns are useful, for different perspectives: preference (RQ2) and adoption (RQ3).

3.2 Evaluation Methodology for REVISAR

Data collection We selected 9 popular GitHub Java projects (Table 1) from a list of previously studied projects [45]. The project selection influences quantity and quality of the discovered patterns. We select mature popular projects with a long history of edits, experienced developers who detect problems during code reviews, and several collaborators (avg. 89.11) with different levels of expertise (low expertise collaborators likely submit pull-requests that need quick fixes). Analyzing many projects is prohibitive given our resources, thus we selected a subset of projects with various sizes/domains. We favored projects containing between 1,000 and 15,000 revisions, to have a sample large enough to identify many patterns but not too big due to the time required to evaluate all revisions. The projects have size ranging from 24,753 to 1,119,579 lines of code. In total, the sample contains 43,113 revisions.

Benchmarks REVISAR found 288,899 single-location edits which were clustered in 110,384 clusters. Of these clusters, 39,104 contained more than one edit—i.e., REVISAR could generalize multiple examples to a single *edit pattern*. The 39,104 edit patterns covered 205,934/288,899 single-location edits (71%). The distribution of these edit patterns is reminiscent of a long-tail one: the most-common edit pattern having 2,706 concrete edits, 0.06% of the edit patterns cover 10% of the concrete edits, and 5.3% of the edit patterns cover 20% of the concrete edits.

We performed the experiments on a PC running Windows 10 with a processor Core i7 and 16GB of RAM. We obtained the revision histories of each repository using JGit[3]. REVISAR took 5 days to analyze the 9 projects (approximately 10 seconds per commit). Most of the time is spent checking out revisions, a process that can be done incrementally for future projects.

Edit pattern sampling To facilitate manual investigation of these edit patterns, we empirically identified a "Goldilocks" cut-offs for edit patterns found in n or more projects that would allow us to inspect each patterns (at $n \geq 1$: 110,384 edit patterns, 2: 1,759, 3: 495, 4: 196; 5: 89, 6: 47, 7: 19, 8: 6, 9: 1). From this distribution, we choose the 495 patterns found in 3+ projects as a reasonable cut-off for subsequent analysis.

Analysis *Phase I—Spurious pattern elimination* To assess the effectiveness of REVISAR, we conducted a filtering exercise to discard spurious edit patterns. Specifically, we discarded edit patterns involving renaming operations—e.g., renaming the variable `obj` to `object`—and edit patterns in which none of the authors could identify a logical rationale behind the edit, typically because the patterns were part of some broader edit sequence—e.g., changing `return true` to `return null`. We eliminated 297 spurious patterns, where 61 of them were renaming operation and, for the rest of them, we could not identify a logically meaningful pattern.

Phase II—Merging duplicate edit patterns Next, we merged edit patterns that represented the same logical quick fix. To do so, we employed a technique of *negotiated agreement* [7] in which the second and fourth authors collaboratively identified and discarded logically duplicate edit patterns within the sample. When there was disagreement about whether two edit patterns were logical duplicates, the authors opted to merge these duplicates, thus penalizing the effectiveness of REVISAR. In other words, this measure is an *upper bound* of the number of duplicates within the edit patterns. We merged 109 duplicated patterns into 17 other patterns.

[3]https://www.eclipse.org/jgit/

Table 1: Projects used to detect edit patterns

Project	Edits	LOC	Revisions
Hive	94,921	1,119,579	11,467
Ant	49,680	137,203	13,790
Guava	28,784	325,902	4,633
Drill	26,173	350,756	2,902
ExoPlayer	20,726	85,305	3,875
Giraph	8,836	99,274	1,062
Gson	4,435	24,753	1,393
Truth	3,857	27,427	1,137
Error Prone	3,200	116,023	2,854
Totals	240,612	2,286,222	43,113

Phase III—Cataloging edit patterns Finally, we classified each of the remaining 495-297-109=89 patterns against the eight Java rule-set from the PMD code analyzer tool (for example, "Performance"). The full rule-set is shown in Table 2. We targeted the PMD rule-set because the PMD developers employed a principled process for designing this rule-set. In particular, a goal of the rule-set was to make the rule-set useful for reporting by third-party tools and techniques.[4] The first and third authors independently mapped the edit patterns to one of the PMD rule-sets. We computed Cohen's Kappa to assess the measure of agreement, and deferred to the first author's judgment to reconcile disagreement. Finally, the first author used online resources, such as Stack Overflow and analyzers' documentation to tag each quick fix as being available or not available in the catalog of an existing tool.

3.3 Developer Survey

We conducted a survey to assess programmers' judgments about a subset of our discovered quick fixes.

Participants We randomly invited 2,000 programmers to participate in our survey through e-mail, collected from author metadata in the commits from 124 popular GitHub Java projects [45], such as Google and Facebook. We received 164 responses (response rate 8.2%). Through demographic questions, 118 participants (72%) reported having 5+ years of Java experience, and also reported using tools to flag code patterns, including IntelliJ (72%), Checkstyle (50%), Sonar (50%), FindBugs (43%), PMD (31%), Eclipse (8%), Error Prone (7%), and others (8%). We did not compensate responses.

Survey protocol To be within responses of 5-10 minutes, we presented programmers with 9 out of the 89 edit patterns using purposive sampling (i.e., we deliberately selected from a design space of candidate edit patterns to balance patterns found in existing

[4]https://github.com/pmd/pmd/wiki/Rule-Categories

Table 2: Description of the categories and frequency

PMD ruleset	Description	*n*
Best Practices	Rules which enforce generally accepted best practices.	19
Code Style	Rules which enforce a specific coding style.	16
Design	Rules which help you discover design issues.	22
Documentation	Rules which are related to code documentation.	5
Error Prone	Rules which detect constructs that are either broken, extremely confusing or prone to runtime errors.	12
Multithreading	Rules which flag issues when dealing with multiple threads of execution.	2
Performance	Rules which flag suboptimal code.	13
Security	Rules which flag potential security flaws.	0
Total		89

analysis tools versus new edit patterns identified). This selection allowed us to verify whether programmers chose patterns found in existing tools as well as to assess their preferences for edit patterns *not* found in current tools.

We presented edit patterns as side-by-side left and right panes, with one pane having the baseline code pattern ("expected bad") and the other with the quick fix version of the code pattern ("expected good"). Each pair was randomized and labeled simply as pattern A and pattern B, so that programmers could not obviously identify the quick fix version of the pattern. To assess if developers preferred the quick fix version of the pattern, we presented a five-point Likert-type item scale: strongly prefer (A) to strongly prefer (B). Programmers were allowed to provide an open-ended comment about their preferences.

Presented edit patterns We presented programmers with the following nine edit patterns:

EP1 (Performance) **Use characters instead of single-character strings.** In Java, we can represent a character both as a `String` or a character. For operations such as appending a value to a `StringBuffer`, representing the value as a character improves performance—e.g., change `sb.append("a")` to `sb.append('a')`. This edit improved the performance in Guava by 10-25% [14].

EP2 (Error Prone) **Prefer string literal in `equals` method.** Invoking `equals` on a `null` variable causes a `NullPointerException`, as discussed. Use `"str".equals(s)` instead of `s.equals("str")`.

EP3 (Performance) **Avoid `FileInputStream` and `FileOutputStream`**. These classes override the `finalize` method. As a result, their objects are only cleaned when the garbage collector performs a sweep [10]. Since Java 7, programmers can use `Files.newInputStream` and `Files.newOutputStream` instead of `FileInputStream` and `FileOutputStream` to improve performance as recommended in this Java JDK bug-report [19].

EP4 (Best practices) **Use the collection `isEmpty` method rather than checking the size.** The method `isEmpty` to check whether a collection is empty is preferred to checking that the size of the collection is zero, as discussed.

EP5 (Multithreading) **Prefer `StringBuilder` to `StringBuffer`.** These classes have the same API, but `StringBuilder` is not synchronized. Since synchronization is rarely used [39], `StringBuilder` offers high performance and is designed to replace `StringBuffer` in single threaded contexts [39].

EP6 (Code Style) **Infer type in generic instance creation.** Since Java 7, programmers can replace type parameters to invoke the constructor of a generic class with the diamond operator (<>) [40] and allow inference of type parameters by the context. This edit ensures the use of generic instead of the deprecated raw types [38]. The benefit of the diamond operator is clarity since it is more concise.

EP7 (Design) **Remove raw types.** Raw types are generic types without type parameters, used in Java versions prior to 5.0. They ensure compatibility with pre-generics code. Since type parameters of raw types are unchecked, unsafe code is caught at runtime [38] and the Java compiler issues warnings for them [38]. Thus, prefer `List<String> a = new ArrayList<>()` to `List<String> a = new ArrayList()`.

EP8 (Error Prone) **Field, parameter, and variable could be `final`.** The `final` modifier can be used in fields, parameters, and local variables to indicate they cannot be reassigned [41]. This edit improves clarity and it helps with debugging since it shows what values will change at runtime. In addition, it allows the compiler and virtual machine to optimize the code [41]. The edit pattern that adds the `final` modifier is included in PMD catalog of rules[5]. IDEs such as Eclipse and NetBeans can be configured to perform this edit automatically on saving.

EP9 (Error Prone) **Avoid using strings to represent paths.** Programmers sometimes use `String` to represent a file system path even though some classes are specifically designed for this task—e.g., `java.nio.Path`. In these cases, it is useful to change the type of the variable to `Path`. First, strings can be combined in an undisciplined way, which can lead to invalid paths. Second, different operating systems use different file separators, which can cause bugs. Since detecting this pattern requires a non-trivial analysis, code analyzers do not include it as a rule. Thus, use `Path path` over `String path`.

Analysis We treated the Likert-type responses as ordinal data and applied a one-sample Wilcoxon signed-rank test to identify statistical differences for each of the nine edit patterns ($\alpha = 0.05$). Specifically, the null hypothesis is that the responses are not statistically different and symmetric around the default value ("it does not matter"). Rejecting the null hypothesis implies that programmers

[5]https://pmd.github.io/

have a non-default preference for one code pattern. Because multiple comparisons can inflate the false discovery rate, we compute adjusted p-values using a Benjamini-Hochberg correction [4]. We present the results for each pair as a net stacked distribution.

3.4 Pull Request Validation

We also submitted PRs to GitHub projects containing the nine quick fixes from our survey.

Project selection From the nine GitHub projects (Table 1), we selected five projects that actively considered PRs (Ant, Error Prone, ExoPlayer, Giraph, and Gson). We supplement these projects with those of four popular code analyzer tools (Checkstyle, PMD, SonarQube, and Spotbugs) with the expectation that reviewers of these pull requests could capably assess the usefulness of the proposed quick fixes. Thus, we selected a total of nine projects to submit PRs.

Pull requests We deliberately submitted PRs manually that applied locally to a single region of code preferable within a single or file to minimize confounds that would be otherwise introduced in large PRs. In total, we submitted 16 PRs across all projects.

Analysis We recorded the status of the PRs as either open (not yet accepted), merged (accepted), or rejected (declined to accept into the project code). We describe these PR submissions through basic descriptive statistics.

4 RESULTS

4.1 How effective is Revisar in identifying quick fixes?

Table 2 shows the identified edit patterns and labels them according to the PMD rule-sets. The discovered patterns covered seven of the eight PMD categories, and only "Security" was not represented. The most common rule-sets—with roughly equal frequencies—were "Design" (22), "Best Practices" (19), "Performance" (13), and "Error Prone" (12). The results suggest that Revisar is effective at discovering quick fixes across a spectrum of rule-sets.

Cohen's κ found "very good" [24] agreement between the raters for these rule-sets ($n = 89$, $\kappa = 0.82$), with disagreement being primarily attributable to whether an edit pattern is "Best Practice" or "Error Prone."

Finally, 57/89 patterns were classified as new ones—i.e., they were not implemented as quick fixes in existing tools.

Revisar could automatically discover 89 edit patterns that covered 7/8 PMD categories. 64% of the discovered patterns did not appear in existing tools.

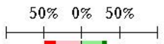
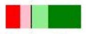
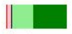
4.2 Do developers prefer Revisar quick fixes?

A summary of the survey results is presented in Table 3. The Wilcoxon signed-rank test identified a significant difference in preference—after Benjamini-Hochberg adjustment—for all nine edit patterns (at $\alpha = 0.05$). Except for EP1, programmers preferred the quick fix version of the code from Revisar.

To understand why programmers rejected EP1, we examined the optional programmer feedback for this edit pattern. We found that although programmers recognized that passing a character would have better performance, "slightly more efficient," and requiring "less overhead," these benefits were not significant enough to outweigh readability or consistency. For example, five programmers reported that since the name of the class is `StringBuffer`, it's more consistent to always pass in a `String`, even if a character would be more efficient. Other programmers reported that always passing in a `String` is just "easier mentally" and requires "less cognitive load."

Programmers preferred the quick fixes suggested by Revisar for eight of the nine edit patterns.

Table 3: Programmer preferences for edit patterns

		Likert Resp. Pct[1]			
Pattern	Adj-p[2]	B	N	QF	Distribution[3]
					
EP1	.01	49%	18%	33%	
EP2	< .001	33%	2%	65%	
EP3	.03	36%	18%	46%	
EP4	< .001	2%	5%	93%	
EP5	< .001	7%	21%	72%	
EP6	< .001	10%	1%	89%	
EP7	< .001	19%	6%	75%	
EP8	< .001	33%	12%	55%	
EP9	< .001	15%	16%	69%	

[1] Likert-type item responses: Strongly prefer or prefer baseline (B), Neutral (N), Strongly prefer or prefer quick fix (QF).
[2] Adjusted p-value after Benjamini-Hochberg correction.
[3] Net stacked distribution removes the Neutral option and shows the skew between baseline and quick fix preferences. 🟥 Strongly prefer baseline, ▨ Prefer baseline, ▧ Prefer quick fix, 🟩 Strongly prefer quick fix.

4.3 Do developers adopt Revisar quick fixes?

Of the 16 PRs we submitted to GitHub projects, seven of these were accepted, four were rejected, and the remaining are open at this writing time (Table 4). SonarQube rejected a PR for EP1, suggesting that changing a `String` to a `Character` is purely pedantic. However, they welcomed additional evidence of the performance benefits and would be willing to reconsider given such evidence. Error Prone programmers indicated that EP2 was generally useful, but not for the particular use case of the PR. SpotBugs rejected the PR for EP5 because a maintainer did not want to unnecessarily make the commit history noisy unless the change was in a performance critical path. Finally, Gson rejected a EP8 PR for adding `final` to a parameter: namely, because their IntelliJ already highlights locals and parameters differently depending on whether they are assigned to. In other words, the programmers already use an alternative means to communicate information about effectively `final` parameters.

Table 4: Pull request submissions to projects on GitHub

Pattern	Accept	(%)	Status[1]
EP1	1	(33%)	🟩 PMD ⬛ Ant 🟥 SonarQube
EP2	2	(67%)	🟩 Ant 🟩 ExoPlayer 🟥 Error Prone
EP3	—	—	⬛ Giraph
EP4	2	(100%)	🟩 Ant 🟩 PMD
EP5	1	(33%)	🟩 CheckStyle ⬛ Ant 🟥 Spotbugs
EP6	1	(100%)	🟩 Giraph
EP7	—	—	⬛ Gson
EP8	0	(0%)	🟥 Gson
EP9	—	—	⬛ Giraph
Total[2]	7 / 11	(63.6%)	

[1] 🟩 Accepted, 🟥 Rejected, ⬛ Open.
[2] Acceptance rate is calculated as accepted pull requests against accepted and rejected pull requests. Open pull requests are not included in this calculation.

Projects accepted 63.6% of the PRs for the REVISAR quick fixes.

5 LIMITATIONS

Each of the three studies have limitations, described here

Evaluation of REVISAR We considered only single-location edit patterns, which are representative of most quick fixes in code analyzers. REVISAR cannot identify dependent patterns—e.g., when both return type of the method and the return statement must change together. Also, we only evaluated Java projects; both the choice of language and projects influences the identified quick fixes. A threat to construct validity is the difficulty to exhaustively determine whether a discovered quick fix is actually novel. To mitigate it, we catalogued popular code analyzers and conducted searches to find quick fixes. Similarly, given that the categorization of quick fixes involves human judgments, our results (Table 2) should be interpreted as useful estimators for REVISAR.

Survey study. The survey employed purposive—that is, non-random—sampling and evaluated only a limited number of quick fixes that do not cover the entire design space of quick fixes. It also lacks some categories (e.g., documentation and security). The documentation category lacks a well-defined edit structure, making it hard to identify relevant quick fixes, besides those in code analyzers. The lack of security quick fixes can be due to the projects selected, since they are mature projects, from large companies, such as Google, which tends to focus on security problem removal. Thus, a threat to external validity is that we should be careful and avoid generalizing the results from this survey to all quick fixes. Moreover, participants in the survey self-reported their experience and may not necessarily have been experts. A construct threat within this study is that programmers are not directly evaluating quick fixes: rather, they are being asked to evaluate two different code snippets—essentially, the input and output to an editor pattern. Responses and explanations for their preferences may have been different had they been explicitly told to evaluate the quick fixes directly.

PR validation A construct validity threat is that PRs are not the typical environment that programmers apply quick fixes. Thus, the acceptance and rejection of PRs are not representative of how programmers would actually apply quick fixes. Despite this limitation, the study validates that discovered quick fixes are adopted by projects, and provides explanation in cases for when they are not.

REVISAR's automation Manual inspection can be costly, an common issue in the field of data mining. All steps of our technique are automated, except for the inspection of the patterns to classify them in new and useful patterns. The applicability of quick fixes in general need to be evaluated, even for quick fixes in code analyers, since their use depends on developers' opinion. In our experiments, we inspected 495 quick fixes in about one day.

REVISAR's scalability The time taken to identify quick fixes depends on the number of projects, revisions, and edits. REVISAR's users can choose the accuracy level. A good accuracy needs more projects, but takes longer to complete. Our technique is not intended to be used *online* (i.e., along with code editing). Thus, patterns can be discovered *offline* and integrated into code analyzers.

6 DISCUSSION

Generating executable rules REVISAR generates AST patterns, but ideally one wants executable quick fixes that can be added to code analyzers. When possible, REVISAR compiles the generated patterns to executable Refaster rules. Refaster [48] is a rule-language used in the code analyzer Error Prone. A Refaster rule is described using (*i*) a before template to pattern-match target locations, and (*ii*) an after template to specify how these locations are transformed, which are similar to the before and after templates τ_i and τ_o used by our rules. In general, our rules cannot be always expressed as Refaster ones. In particular, Refaster cannot describe edit patterns that require AST node types. For instance, the edit pattern in Figure 4 requires knowing that an AST node is a `StringLiteral` and this AST type cannot be inspected in Refaster. In addition, Refaster can only modify expressions that appear inside a method body—e.g., Refaster cannot modify global field declarations.

Programmers consider trade-offs when applying quick fixes Programmers' feedback within our survey suggests trade-offs that programmers consider when making judgments about applying quick fixes. For example, for EP1 some programmers preferred the version of the code with worse performance primarily because they valued *consistency* and *reduction in cognitive load* over what they felt was relatively small performance improvements.

When programmers significantly preferred quick fixes, they carefully evaluated the trade-offs. For example, consider EP2, in which the quick fix suggests using the `equals method` on string literals to prevent an exception. Programmers recognized this benefit but also argued that the baseline version had better *readability*. As one programmer notes, when given a variable, they felt it more natural to say, "if variable equals value" than "if value equals variable."

Programmers also indicated *unfamiliarity* with new language features as a reason to avoid quick fixes (e.g., in EP3). One programmer noted that newer APIs embed "experience about the shortcomings of the old API" but they were also hesitant to use this version of the code without understanding what the shortcomings actually were.

Finally, using the diamond operator (<>) in EP6 makes the code simpler, concise, and more readable. Nevertheless, 19% of participants still preferred/strongly preferred the less concise baseline code version. One programmer suggested *compatibility* with old versions as a reason for this decision.

Thus, even when automated techniques such as Revisar discover useful quick fixes, the feedback from our survey suggests that it is also important to provide programmers with rationale for why and when the quick fix should be applied. A first-step towards providing an initial rationale can be to situate quick fixes within an existing taxonomy, as we did with our discovered quick fixes in Table 2.

Barriers to accepting PRs Although our survey indicated programmers preferred Revisar quick fixes, maintainers can decline the corresponding PRs. In our PR study, the maintainers' comments suggested reasons for declining a quick fix, even when they recognized the fixes would be *generally* useful. For instance, the maintainers of SonarQube declined incorporating the fixes because it would make the commit history more noisy. Spotbugs was concerned about adopting them without sufficient testing because the fixes might behave unexpectedly in different JVM. Other projects like Error Prone have adopted conventions across their entire code base. Unless these quick fixes are applied universally across the project, such inertia makes it unlikely that these projects would adopt a one-off fix—for example, EP2.

Our analysis suggests that *when* and *how* a quick fix is surfaced to the developer is important to its acceptance. It is possible these maintainers would have applied these rejected quick fixes had they been revealed *as* they were writing code, rather than after the fact.

7 RELATED WORK

Mining quick fixes Cayres et al. [8] show a systematic review of the techniques to suggest fixes from histories. Brown et al. [5] learn token-level syntactic transformations from code commits to generate mutations for mutation testing. Unlike Revisar, they can only mine token level transformations over a predefined set of syntactic constructs and cannot unify across multiple concrete edits. Negara et al. [34] mine code interactions directly from the IDE to detect quick fixes and find 10 new refactoring patterns. Revisar mines repositories. Hora et al. [17] create quick fixes by automatically mining repositories from a system history to detect change rules. Instead, Revisar mines a set of systems. Hora et al. [16] observe defects regarding generic and domain-specific checkers by mining defective lines from commits messages and correlated them with those from code analyzers. Revisar is not focused only on defects and mines quick fixes across repositories. Other tools [29, 46] mine fine-grained repair templates from StackOverflow and the Defect4j bug data-set. Liu et al. [27] collect a set of fixes from repositories and apply machine learning techniques to ranking them, based on recurrent code fixes. Janke and Mäder [18] capture the relational context of individual edit patterns from repositories. Li et al. [25] investigate changes that can significantly influence systems and their uses. Neubig and Gaunt [49] represent edits both in natural language and code. Marcilio et al. [31] suggest corrections for well-known problems of code analyzers. In summary, Revisar differs from prior since we (*i*) mined quick fixes in a sound and complete fashion using a rich syntax of edit patterns, (*ii*) assessed the quality of the learned patterns and their quick fixes.

Learning transformations from examples Several techniques use examples to learn quick fixes for refactoring [1, 33, 43], clone removal [32], defect removal [21], learn to fix command-line errors [9], and complete code [13, 15]. All these techniques rely on user-given examples that describe the same intended transformation or on curated labeled data. This extra information allows the tools to perform more informed types of rule extraction. Instead, Revisar uses fully unsupervised learning and receives concrete edits as input that may or may not describe useful transformations.

Program repair Some program repair tools learn useful fixing strategies by mining curated sets of bug fixes [36], user interactions with a debugger [20], human-written patches [22, 30], and bug reports [26]. FixMiner [23] identifies relevant and actionable quick fixes based on atomic edits, and PARFixMiner [23] performs quick fixes. DevReplay builds regular expression rules from two revision history to help applying quick fixes [47]. TBar [28] uses templates to apply quick fixes to program bugs. Sakkas et al. [44] use a data-driven strategy to providing feedback for type-errors. Koyuncu et al. [27] use fixes patterns of code analyzers to generate patches. Renggli et al. [42] check domain-specific rules, and implement and automate removal of violations identified by these rules. All these tools either rely on a predefined set of patches or learn patches from supervised data. Instead, Revisar analyzes unsupervised sets of concrete edits and uses a sound and complete technique for mining a well-defined family of edit patterns. Moreover, Revisar learns arbitrary quick fixes that can improve code quality not just ones used to repair buggy code. Since we do not have a notion of correct edit pattern we also analyze the usefulness of the learned quick fixes through a comprehensive evaluation and user study, a optional component in transformations used in program repair.

8 CONCLUSION

We presented Revisar, a technique to learn quick fixes from repositories. Revisar (*i*) identifies edits by comparing consecutive revisions in repositories, (*ii*) clusters edits into sets that can be abstracted into the same edit pattern, and we used Revisar to mine quick fixes from nine popular GitHub projects and Revisar successfully learned 89 edit patterns that appeared in more than three projects. To assess whether programmers would like to apply these quick fixes, we surveyed 164 programmers showing 9 of our quick fixes. Overall, programmers supported 89% them. We also issued PRs in various repositories and 63.6% were accepted so far.

The results have several implications for toolsmiths: (*i*) Revisar can be used to efficiently collect patterns and their usages in actual repositories and enable toolsmiths to make informed decisions about which quick fixes to prioritize based on patterns programmers actually apply in practice. (*ii*) Revisar allows toolsmiths to discover new quick fixes, without needing their users to explicitly submit quick fix suggestions. (*iii*) the results suggest several logical and useful extensions to aid toolsmiths—e.g., supporting more complex patterns from code analyzers but are currently beyond the capabilities of Revisar and designing techniques for automatically extracting executable quick fixes from mined patterns.

REFERENCES

[1] J. Andersen and J. L. Lawall. 2008. Generic Patch Inference. In *Proceedings of the 23rd International Conference on Automated Software Engineering* (L'Aquila, AQ, Italy) *(ASE '08)*. IEEE Computer Society, Washington, DC, USA, 337–346.

[2] A. Baumgartner and T. Kutsia. 2017. Unranked second-order anti-unification. *Information and Computation* 255, 2 (2017), 262 – 286.

[3] A. Baumgartner, T. Kutsia, J. Levy, and M. Villaret. 2017. Higher-Order Pattern Anti-Unification in Linear Time. *Journal of Automated Reasoning* 58, 1 (2017), 293–310.

[4] Yoav Benjamini and Yosef Hochberg. 1995. Controlling the False Discovery Rate: A Practical and Powerful Approach to Multiple Testing. *Journal of the Royal Statistical Society. Series B (Methodological)* 57, 1 (1995), 289–300.

[5] David Bingham Brown, Michael Vaughn, Ben Liblit, and Thomas Reps. 2017. The Care and Feeding of Wild-caught Mutants. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE '17)*. ACM, New York, NY, USA, 511–522.

[6] Peter Bulychev and Marius Minea. 2009. An evaluation of duplicate code detection using anti-unification. In *Proceedings of the 3rd International Workshop On Software Clones* (Kaiserslautern, Germany) *(IWSC '09)*. Fraunhofer IESE, Kaiserslautern, Germany, 1–6.

[7] J. Campbell, C. Quincy, J. Osserman, and O. Pedersen. 2013. Coding in-depth semistructured interviews. *Sociological Methods & Research* 42, 3 (2013), 294–320.

[8] Leandro Ungari Cayres, Bruno Santos de Lima, and Rogério Eduardo García. 2019. Learning and Suggesting Source Code Changes from Version History: A Systematic Review. *arXiv: Software Engineering* (2019), 1–15.

[9] Loris D'Antoni, Rishabh Singh, and Michael Vaughn. 2017. NoFAQ: Synthesizing Command Repairs from Examples. In *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE'17)*. ACM, New York, NY, USA, 582–592.

[10] DZone. 2021. FileInputStream / FileOutputStream Considered Harmful. At https://dzone.com/articles/fileinputstream-fileoutputstream-considered-harmful. Accessed in 2021, July 14.

[11] W. S. Evans, C. W. Fraser, and F. Ma. 2007. Clone Detection via Structural Abstraction. In *Proceedings of 14th Working Conference on Reverse Engineering* (Vancouver, BC, Canada) *(WCRE '07)*. IEEE, Piscataway, NJ, USA, 150–159.

[12] J. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus. 2014. Fine-grained and Accurate Source Code Differencing. In *Proceedings of the 29th International Conference on Automated Software Engineering* (Vasteras, Sweden) *(ASE '14)*. ACM, New York, NY, USA, 313–324.

[13] Mark Gabel and Zhendong Su. 2010. A Study of the Uniqueness of Source Code. In *Proceedings of the 18th International Symposium on Foundations of Software Engineering* (Santa Fe, USA) *(FSE '10)*. ACM, New York, NY, USA, 147–156.

[14] Google. 2012. Guava. At https://docs.oracle.com/javase/9/docs/api/java/lang/Float.html. Accessed in 2021, July 14.

[15] Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. 2012. On the Naturalness of Software. In *Proceedings of the 34th International Conference on Software Engineering* (Zurich, Switzerland) *(ICSE '12)*. IEEE, Piscataway, NJ, USA, 837–847.

[16] A. Hora, N. Anquetil, S. Ducasse, and S. Allier. 2012. Domain specific warnings: Are they any better?. In *Proceedings of the 28th International Conference on Software Maintenance* (Trento, Italy) *((ICSM' 12)*. IEEE, Piscataway, USA, 441–450.

[17] André Hora, Nicolas Anquetil, Anne Etien, Stéphane Ducasse, and Marco Túlio Valente. 2015. Automatic detection of system-specific conventions unknown to developers. *Journal of Systems and Software* 109, 1 (2015), 192–204.

[18] M. Janke and P. Mader. 2020. Graph Based Mining of Code Change Patterns from Version Control Commits. *Transactions on Software Engineering* 1, 1 (2020), 1–16.

[19] Java JDK. 2021. JDK Bug System. At https://bugs.openjdk.java.net/browse/JDK-8187325. Accessed in 2021, July 14.

[20] D. Jeffrey, M. Feng, N. Gupta, and R. Gupta. 2009. BugFix: A learning-based tool to assist developers in fixing bugs. In *Proceedings of the 17th International Conference on Program Comprehension (ICPC '09)*. IEEE, Piscataway, USA, 70–79.

[21] M. Kessentini, W. Kessentini, H. Sahraoui, M. Boukadoum, and A. Ouni. 2011. Design Defects Detection and Correction by Example. In *Proceedings of the 19th International Conference on Program Comprehension (ICPC '11)*. IEEE, Piscataway, NJ, USA, 81–90.

[22] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 2013. Automatic Patch Generation Learned from Human-written Patches. In *Proceedings of the 35th International Conference on Software Engineering* (San Francisco, USA) *(ICSE '13)*. IEEE, Piscataway, NJ, USA, 802–811.

[23] A. Koyuncu, K. Liu, T. Bissyandé, D. Kim, J. Klein, M. Monperrus, and Y. Le Traon. 2020. FixMiner: Mining relevant fix patterns for automated program repair. *Empirical Software Engineering* 25, 3 (2020), 1980–2024.

[24] J. Richard Landis and Gary G. Koch. 1977. The Measurement of Observer Agreement for Categorical Data. *Biometrics* 33, 1 (1977), 159–174.

[25] Daoyuan Li, Li Li, Dongsun Kim, Tegawendé F. Bissyandé, David Lo, and Yves Le Traon. 2019. Watch out for this commit! A study of influential software changes. *Journal of Software: Evolution and Process* 31, 12 (2019), 1–25.

[26] C. Liu, J. Yang, L. Tan, and M. Hafiz. 2013. R2Fix: Automatically Generating Bug Fixes from Bug Reports. In *Proceedings of the 6th International Conference on Software Testing, Verification and Validation* (Neumunster Abbey, Luxembourg, French) *(ICST '13)*. IEEE, Piscataway, NJ, USA, 282–291.

[27] K. Liu, D. Kim, T. Bissyande, S. Yoo, and Y. Traon. 2018. Mining Fix Patterns for FindBugs Violations. *Transactions on Software Engineering* 47, 1 (2018), 165–188.

[28] K. Liu, A. Koyuncu, D. Kim, and T. Bissyandé. 2019. TBar: Revisiting Template-Based Automated Program Repair. In *Proceedings of the 28th International Symposium on Software Testing and Analysis* (Beijing, China) *(ISSTA' 19)*. ACM, New York, NY, USA, 31–42.

[29] X. Liu and H. Zhong. 2018. Mining stackoverflow for program repair. In *Proceedings of the 25th International Conference on Software Analysis, Evolution and Reengineering* (Campobasso, Italy) *(SANER '18)*. IEEE, Piscataway, USA, 118–129.

[30] Fan Long and Martin Rinard. 2016. Automatic Patch Generation by Learning Correct Code. *Proceedings of the 43rd Symposium on Principles of Programming Languages* 51, 1 (2016), 298–312.

[31] Diego Marcilio, Carlo A. Furia, Rodrigo Bonifácio, and Gustavo Pinto. 2020. SpongeBugs: Automatically generating fix suggestions in response to static code analysis warnings. *Journal of Systems and Software* 168, 1 (2020), 1–20.

[32] Na Meng, Lisa Hua, Miryung Kim, and Kathryn S. McKinley. 2015. Does Automated Refactoring Obviate Systematic Editing?. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1* (Florence, Italy) *(ICSE '15)*. IEEE, Piscataway, NJ, USA, 392–402.

[33] Na Meng, Miryung Kim, and Kathryn S. McKinley. 2013. LASE: Locating and Applying Systematic Edits by Learning from Examples. In *Proceedings of the 35th International Conference on Software Engineering* (San Francisco, USA) *(ICSE '13)*. IEEE, Piscataway, NJ, USA, 502–511.

[34] Stas Negara, Mihai Codoban, Danny Dig, and Ralph E. Johnson. 2014. Mining Fine-grained Code Changes to Detect Unknown Change Patterns. In *Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) *(ICSE' 14)*. ACM, New York, NY, USA, 803–813.

[35] Hoan Anh Nguyen, Anh Tuan Nguyen, Tung Thanh Nguyen, Tien N. Nguyen, and H. Rajan. 2013. A study of repetitiveness of code changes in software evolution. In *Proceedings of the 28th International Conference on Automated Software Engineering* (Silicon Valley, USA) *(ASE '13)*. IEEE, Piscataway, NJ, USA, 180–190.

[36] Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H. Pham, Jafar Al-Kofahi, and Tien N. Nguyen. 2010. Recurring Bug Fixes in Object-oriented Programs. In *Proceedings of the 32nd International Conference on Software Engineering* (Cape Town, South Africa) *(ICSE '10)*. ACM, New York, NY, USA, 315–324.

[37] Oracle. 2021. ConcurrentSkipListSet. At https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ConcurrentSkipListSet.html. Accessed in 2021, July 14.

[38] Oracle. 2021. Raw Types. At https://docs.oracle.com/javase/tutorial/java/generics/rawTypes.html. Accessed in 2021, July 14.

[39] Oracle. 2021. StringBuilder. At https://docs.oracle.com/javase/8/docs/api/java/lang/StringBuilder.html. Accessed in 2021, July 14.

[40] Oracle. 2021. Type Inference for Generic Instance Creation. At https://docs.oracle.com/javase/7/docs/technotes/guides/language/type-inference-generic-instance-creation.html. Accessed in 2021, July 14.

[41] Java Practices. 2021. Use final liberally. At http://www.javapractices.com/topic/TopicAction.do?Id=23. Accessed in 2021, July 14.

[42] L. Renggli, S. Ducasse, T. Gîrba, and O. Nierstrasz. 2010. Domain-Specific Program Checking. In *Proceedings of the 48th International Conference Objects, Models, Components, Patterns* (Málaga, Spain) *(TOOLS' 10)*, J. Vitek (Ed.). Springer, Berlin, Heidelberg, 213–232.

[43] R. Rolim, G. Soares, L. D'Antoni, O. Polozov, S. Gulwani, R. Gheyi, R. Suzuki, and B. Hartmann. 2017. Learning Syntactic Program Transformations from Examples. In *Proceedings of the 39th International Conference on Software Engineering* (Buenos Aires, Argentina) *(ICSE '17)*. IEEE, Piscataway, NJ, USA, 404–415.

[44] G. Sakkas, M. Endres, B. Cosman, W. Weimer, and R. Jhala. 2020. Type Error Feedback via Analytic Program Repair. In *Proceedings of the 41st Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. ACM, New York, NY, USA, 16–30.

[45] Danilo Silva, Nikolaos Tsantalis, and Marco Tulio Valente. 2016. Why We Refactor? Confessions of GitHub Contributors. In *Proceedings of the 24th International Symposium on Foundations of Software Engineering* (Seattle, WA, USA) *(FSE 2016)*. Association for Computing Machinery, New York, NY, USA, 858–870.

[46] Victor Sobreira, Thomas Durieux, Fernanda Madeiral, Martin Monperrus, and Marcelo A. Maia. 2018. Dissection of a Bug Dataset: Anatomy of 395 Patches from Defects4J. In *Proceedings of the 25th International Conference on Software Analysis, Evolution and Reengineering (SANER '18)*. IEEE, Piscataway, USA, 130–140.

[47] Y. Ueda, T. Ishio, A. Ihara, and K. Matsumoto. 2020. DevReplay: Automatic Repair with Editable Fix Pattern. *ArXiv: Software Engineering* (2020), 1–15.

[48] Louis Wasserman. 2013. Scalable, Example-based Refactorings with Refaster. In *Proceedings of the 6th Workshop on Refactoring Tools* (Indianapolis, USA) *(WRT '13)*. ACM, New York, NY, USA, 25–28.

[49] Pengcheng Yin, Graham Neubig, Miltiadis Allamanis, Marc Brockschmidt, and Alexander L. Gaunt. 2019. Learning to Represent Edits. *arXiv: Software Engineering* (2019), 1–22.