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.

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.

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 about it. Despite its usefulness, a quick fix rule for this edit pattern

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-9061-3/21/09... $15.00
https://doi.org/10.1145/3474624.3474650
We also sent Pull Requests (PRs) applying this quick fix to other parts of the code in the Apache Ant and Google ExoPlayer projects. This paper makes the following contributions:

- For example, useful ones and add the corresponding quick fixes to code analyzers. For example, the edit pattern in Figure 1 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 tool that analyzes the differences between the ASTs of consecutive revisions ($\mathcal{R}_i$ and $\mathcal{R}_{i+1}$) and uses a Tree Edit Distance (TED) algorithm to identify a set of tree edits $\{e_1, e_2, \ldots, e_l\}$ that when applied to the AST $\mathcal{R}_i$ corresponding to $\mathcal{R}_i$ yields the AST $\mathcal{R}_{i+1}$ corresponding to $\mathcal{R}_{i+1}$. 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 edits. To give an example of edits computed by GumTree, let’s 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.

## 2 REVISAR

We now describe REVISAR, our technique for learning quick fixes from code repositories. Given repositories as input, REVISAR:

- identifies concrete code edits by comparing pairs of consecutive revisions (§2.1),
- 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 $\mathcal{R} = \{R_1, \ldots, R_n\}$ where each $R_i$ is a revision history $R_1R_2 \cdots R_k$ 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 $\mathcal{R}_i$ of $r_i$ and $r_{i+1}$ and uses a Tree Edit Distance (TED) algorithm to identify a set of tree edits $\{e_1, e_2, \ldots, e_l\}$ that when applied to the AST $r_i$ corresponding to $r_i$ yields the AST $r_{i+1}$ 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.
look at the first two lines in Figure 1a. Figure 3 illustrates the
ASTs corresponding to `args[1].equals("--launchdiag")` and
"--launchdiag".equals(args[1]), respectively.

![Diagram](image-url)

**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, 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[1].

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[1].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 `e1` and `e2` share the same component if
(i) the two nodes `x1` and `x2` modified by `e1` and `e2` have the same
parent, or (ii) the `x1` (resp. `x2`) is the parent of `x2` (resp. `x1`). For
instance, the previously shown edits are associated to two nodes `x1 =
"--launchdiag"` and `x2 = 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 `{R1, . . . , Rn}`, outputs a set of concrete edits
`{(i1, o1), . . . , (ik, ok)}`.

### 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 `{(i1, o1) . . . (in, on)}` 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 `o_k` be
the substitution for the result of the anti-unification and let `a_k(τ)`
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 `a_k(τ)`. The final
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[1].equals("--launchdiag")`. Upon receiving a
new tree `args[1].equals("--noclasspath")`, anti-unifying the
two trees yields the template `args[1].equals(τ)` with substitutions
`a_1 = {?\_1="--launchdiag"}` and `a_2 = {?\_2="--noclasspath"}`. We have concrete nodes "--launchdiag" and "--noclasspath"
each of size one, and a single hole `?i`. The cost will be $2 = 1$. The
final cost of adding an edit to a cluster is the sum of the cost to
anti-unify `τ1` and `i_k` and the cost to anti-unify `τ2` 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-caps` [2, 11, 35], a
technique for identifying repetitive edits. Given a number `d ≥ 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[1].equals("--launchdiag")`. A
`1-cap` replaces all the nodes at depth one with holes. For our example,
`args[1].equals("--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) . . . (in, on)}`—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 `τ = τ_1 → τ_2` with two components: (i) the
template `τ_1`, which is used to decide whether a subtree `t` in the code
can be transformed using the rule $r$, (ii) the template $\tau_0$, which describes how the tree matching $\tau_1$ should be transformed by $r$.

A template $\tau$ is an AST where leaves can also be holes (variables) and a tree $t$ matches the template $\tau$ if there exists a way to assign concrete values to the holes in $\tau$ and obtain $t$—denoted $t \in L(\tau)$. Given a template $\tau$ over a set of holes $H$, we use $\alpha$ to denote a substitution from $H$ to concrete trees and $\alpha(\tau)$ to denote the application of the substitution $\alpha$ to the holes in $\tau$. Figure 4 shows the first two concrete edits from Figure 1a and the templates $\tau_1$ and $\tau_0$, describing the edit pattern obtained from these examples. Here, the template $\tau_1$ matches any expression calling the method $\text{equals}$ with first argument $\text{args}[i]$ and any possible second argument. The two substitutions $\alpha_1 = \{?_1 = "--launchdiag"\}$ and $\alpha_2 = \{?_1 = "--noclasspath"\}$ yield the expressions $\alpha_1(\tau_1) = \text{args}[i].\text{equals}("--launchdiag")$ and $\alpha_2(\tau_1) = \text{args}[i].\text{equals}("--noclasspath")$, respectively. The template $\tau_0$ is similar to $\tau_1$ and note that the hole $?_1$ appearing in $\tau_0$ is the same as the one appearing in $\tau_1$.

\[
\begin{align*}
\text{args}[i].\text{equals}("--launchdiag") & \rightarrow "--launchdiag".\text{equals}([i]) \\
\text{args}[i].\text{equals}("--noclasspath") & \rightarrow "--noclasspath".\text{equals}([i])
\end{align*}
\]

**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 $r = \tau_1 \rightarrow \tau_0$, is consistent with $S$ if (i) the set of holes in $\tau_0$ is a subset of the set the holes appearing in $\tau_1$ (ii) for every $(i_k, o_k)$ there exists a substitution $\alpha_k$ such that $\alpha_k(\tau_1) = i_k$ and $\alpha_k(\tau_0) = o_k$.

Our goal now is to compute a template $\tau_0$ such that every AST $t_i$ can match the template $\tau_i$—i.e., $t_i \in L(\tau_i)$. 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., $?_1.\text{equals}(?_2)$ is more general than $\text{arg}[i].\text{equals}(?_1)$. 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 $x.\text{equals}("abc")$, we will obtain the desired, more general template $?_1.\text{equals}(?_2)$.

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 $\tau$ for which there exist substitutions $\alpha_1$ and $\alpha_2$ 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 $\tau_1$ and $\tau_0$ 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 $\tau_1$ and the template for the output $\tau_0$. However, Revisar needs to analyze whether these templates describe an edit pattern $r = \tau_1 \rightarrow \tau_0$—i.e., whether there is a way to map the holes of $\tau_1$ to the ones of $\tau_0$. This mapping can be computed by finding, for every hole $?_2$ in $\tau_0$, a hole $?_1$ in $\tau_1$ 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 $\tau_1$ and $\tau_0$, 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 $\tau_1$ is incompatible with the substitution for $?_2$ in $\tau_0$ 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 $(\text{args}[i].\text{equals}("--launchdiag"), "$--launchdiag".\text{equals}([i]))$ and $(\text{args}[i].\text{equals}(!\text{IgnoreCase}("--launchdiag"), "$--launchdiag".\text{equals}(!\text{IgnoreCase}([i])))$ should not be in the same cluster since they are applied to the equals and equalsIgnoreCase methods, respectively.

\[
\begin{align*}
\text{args}[i].\text{equals}("--launchdiag") & \rightarrow "--launchdiag".\text{equals}([i]) \\
\text{args}[i].\text{equals}("--noclasspath") & \rightarrow "--noclasspath".\text{equals}([i])
\end{align*}
\]

**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 $r = \tau_1 \rightarrow \tau_0$ consistent with $S$ if and only if some edit pattern $r' = \tau_1' \rightarrow \tau_0'$ consistent with $S$ exists.

### 3 METHODOLOGY

In this section, we describe our evaluation methodology.

#### 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?

\footnote{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 rule sets. 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. 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 pattern (at n ≥ 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.

Phase III—Cataloging edit patterns Finally, we classified each of the remaining 495-297-109=89 patterns against the eight Java rule-sets 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

---

1 https://www.eclipse.org/jgit/

2 https://github.com/pmd/pmd/wiki/Rule-Categories
Table 2: Description of the categories and frequency

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

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], StringBuffer 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 re-assigned [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. 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 (α = 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
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 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 k 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 ($\alpha = 0.05$). 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.

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.

Projects accepted 63.6% of the PRs for the Revisar quick fixes.
Responses and explanations for their preferences may have been snippets—essentially, the input and output to an editor pattern. 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 what they felt was relatively small performance improvements.

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 influence 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 cataloged 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 analyzers, 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 τi and τp 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 String Literal 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


