Understanding Mirror Bugs in Multiple-Language Projects

YE TANG, HONGHAO CHEN, ZHIXING HE, and HAO ZHONG*, Shanghai Jiao Tong University, China

As software is widely used in daily life, bugs can introduce catastrophic consequences. Researchers have conducted empirical studies to delve into bug characteristics, exploring topics such as buggy locations, symptoms, causes, and repair patterns. To attract users, many applications have implementations in different languages. If an implementation has a bug, other implementations can have similar bugs. In this paper, we term cross-language clone bugs as "mirror bugs". Understanding mirror bugs is crucial, as they offer insights into broader bug patterns. Still, no prior study has explored mirror bugs, leaving several research questions unanswered. For example, can bug fixes in one language help detect and repair bugs in other languages? Is a bug's patch useful for addressing its mirror bugs?

To address these questions, we conducted the first empirical study analyzing mirror bugs. Our investigation focused on 638 bugs from four projects, implemented in both Java and C#. Our study presents answers to four interesting research questions. For example, some programmers actively fix mirror bugs even without tool support. Consequently, there is a timely need for tools that assist in detecting mirror bugs. Following this insight, we manually identified and fixed 9 new mirror bugs, with 5 already accepted by programmers.

ACM Reference Format:
Ye Tang, Honghao Chen, Zhixing He, and Hao Zhong. 2024. Understanding Mirror Bugs in Multiple-Language Projects. *ACM Trans. Softw. Eng. Methodol.* 1, 1 (July 2024), 25 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Software bugs have caused catastrophic consequences and huge monetary losses [93]. For example, as reported by Wong *et al.* [94], the bugs in the ERP system of Hewlett-Packard caused a $160 million loss, and the bugs in the purchasing system of Ford Motor Co. caused a $400 million loss. To understand the characteristics of software bugs, researchers [68, 80, 103] have conducted various empirical studies, and most of these studies analyze bugs whose source files are implemented in a single programming language. These studies analyze bugs in different types of applications from different perspectives. In particular, some studies [68, 80, 97, 103] report that many bugs appear in clones. Following this insight, researchers [64, 109] have proposed approaches to detect bugs in similar code fragments.

To attract more users, many projects are implemented in multiple programming languages. For instance, Similarweb [35] is a company that specializes in analyzing app markets. Its customers include LinkedIn [36] and other widely-used applications. According to its report [37], 100% of American mobile applications have both iOS and Android implementations, and the percentage of global mobile applications is 37%. Even if a project is implemented in a single language, outsiders can

*Manuscript received July, 2024.
(Corresponding author: Hao Zhong.)

Authors' address: Ye Tang, tangye_22@sjtu.edu.cn; Honghao Chen, chenhonghao@sjtu.edu.cn; Zhixing He, chandlr@sjtu.edu.cn; Hao Zhong, zhonghao@sjtu.edu.cn, Shanghai Jiao Tong University, Shanghai, China.

1049-331X/2024/7-ART $15.00
https://doi.org/XXXXXXX.XXXXXXX

re-implement the project in other languages. For example, `LocationTech` implements, Java Topology Suite (JTS) [17], a library that provides geometric functions. Meanwhile, outsiders implement, Net Topology Suite [18], a C# correspondence of `JTS`. Many projects are initially implemented in a language, and are ported to other languages [19, 20]. As a result, these projects can have many similar code fragments in different languages [56, 69, 91]. Like bugs in clones, the similar code fragments in different languages can have similar bugs. In this paper, we call such bugs mirror bugs for short. A study on mirror bugs can motivate the research on corresponding detection techniques.

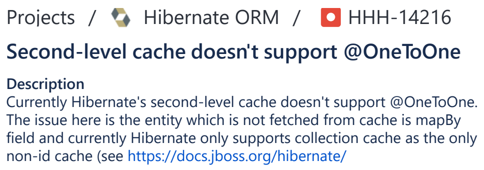
Compared with bugs in clones of the same programming language, language features provide intriguing elements for the studies of mirror bugs. For example, memory leaks are less found in Java code than in C code, since Java provides the garbage collection mechanism to manage memory resources. As another example, two programming languages can provide APIs with subtle differences [107]. The differences can cause mirror bugs that will not appear in clone bugs. To deepen the understanding of bugs, in our ICSE 2024 poster paper [55], we conduct the first empirical study on mirror bugs. In this paper, we report the distribution of mirror bugs in real projects, and introduce our work plans, *e.g.*, analyzing why some bugs have no mirror bugs. We have completed the planned study, and our study explores the following research questions:

- **RQ1. How many mirror bugs are there?**
  Motivation: The answers are useful for increasing the awareness of mirror bugs.
  Answer: Finding 1 shows that in total, we already found 15.0% of bugs in both Java and C# implementations. As the implementations in two languages have many differences during their independent evolution, the percentage cannot be ignored, and can motivate many research topics on mirror bugs.
- **RQ2. How many mirror bugs are fixed?**
  Motivation: The answers are useful for understanding the repairs of mirror bugs.
  Answer: Finding 2 shows that 65.6% of our mirror bugs are fixed. In particular, among the 63 fixed bugs, 48 (76.2%) bugs were identified by their programmers. Among the 33 unfixed bugs, Finding 3 shows that we have produced the symptoms of 27 (81.8%) bugs. Finding 4 shows that detecting mirror bugs is beneficial when the other sides have many bug reports.
- **RQ3. Why do some bugs have no mirror bugs?**
  Motivation: The answers are useful for understanding the differences between projects.
  Answer: For bugs that are not cross-languages, Finding 5 shows that 71.0% of them have no corresponding buggy locations in other implementations. Finding 6 shows that the other causes include language-specific problems (20.3%) and other minor factors (8.7%).

Our study enriches the common knowledge with surprising findings. For example, some researchers believe that mirror bugs are few. As they expected, in `Lucene` and `Hibernate`, mirror bugs are around 10% of the sampled bugs. However, to their surprise, we find around 60% sampled bugs from `JTS` are mirror bugs. Indeed, programmers from `NTS` even actively learn the fixed bugs of `JTS`. If programmers and researchers pay more attention to mirror bugs like `NTS` programmers, they can detect more mirror bugs. In contrast, we find no mirror bugs from `Log4j2`, since `Log4net` is still equivalent to the old version, `Log4j`. We discuss the significance of our findings in Section 5.

Our study is not limited to only suggestions, but are actionable in real development. For example, a major interpretation of our findings is that it is feasible to detect new mirror bugs by learning known bugs. To explore whether this vision works in real development, in Section 6, we try to fulfill this vision on our dataset by answering the following two research questions:

- **RQ4. Can we repair new mirror bugs by manually learning known bugs?**
  Motivation: The answers are useful for understanding the potential of repairing new mirror bugs.

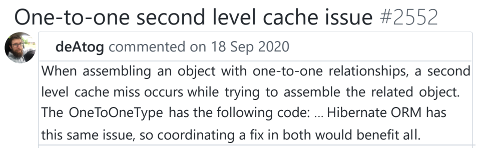

Projects / Hibernate ORM / HHH-14216

Second-level cache doesn't support @OneToOne

Description

Currently Hibernate's second-level cache doesn't support @OneToOne. The issue here is the entity which is not fetched from cache is mapBy field and currently Hibernate only supports collection cache as the only non-id cache (see https://docs.jboss.org/hibernate/

(a) HHH-14216 [4]


One-to-one second level cache issue #2552

deAtog commented on 18 Sep 2020

When assembling an object with one-to-one relationships, a second level cache miss occurs while trying to assemble the related object. The OneToOneType has the following code: ... Hibernate ORM has this same issue, so coordinating a fix in both would benefit all.

(b) NHibernate#2552 [5]

```
public Object assemble (...) ... {
− return resolve ( session . getContextEntityIdentifier (owner), session , owner);
+ Serializable id = ( Serializable ) getIdentifierType ( session ) . assemble(oid , session , null ) ;
+ if ( id == null ) {return null ;}
+ return resolveIdentifier (id , session ) ;
}...
```

(c) The patch for HHH-14216 [6]

```
public override object Assemble (...) {
− return ResolveIdentifier ( session . GetContextEntityIdentifier (owner), session , owner);
+ object id = GetIdentifierType ( session ) . Assemble(cached, session , null ) ;
+ if (id == null ) {return null ;}
+ return ResolveIdentifier (id , session ) ;
}...
```

(d) The patch for NHibernate#2552 [7]

Fig. 1. An example mirror bug.

Answer: We manually identify and repair new bugs from our already analyzed bugs. In total, we have fixed 9 new mirror bugs according to how their corresponding bugs in the other language are fixed. Among them, 5 patches are accepted by their programmers. The results confirm that it is feasible to detect previously unknown mirror bugs, and the knowledge of repairing a bug is useful to repair its mirror bugs.

- **RQ5: What are the challenges if researchers automate the process?**
  Motivation: The answers are useful for understanding the potential and challenges of automatically repairing mirror bugs.
  Answer: We conduct a pilot study on 9 mirror bugs with bug reports on both sides. We find that template-based approaches have the potential of repairing 5 mirror bugs if templates are carefully modified. The remaining 4 bugs could not be fixed by template-based approaches, but mirror bugs provide useful hints to implement their new patches.

In summary, this paper makes the following contributions:

- **An extended empirical study for mirror bugs.** We conducted the first empirical study on mirror bugs [55] and extend this study with extra research questions and detailed analysis. For instance, we construct the first classification framework for mirror bugs in this manuscript.
- **A dataset for mirror bugs.** We present the first publicly accessible dataset of mirror bugs, which comprises 638 bugs from four real-world projects implemented in both Java and C#.
- **Manually fixed mirror bugs.** We make the first attempt to detect and repair new mirror bugs by learning from known bugs. To date, we have identified 9 new mirror bugs and successfully repaired all of them, 5 of which have been confirmed by developers.
- **Lessons for automated program repair.** We conduct a pilot study to explore the feasibility and challenges of repairing mirror bugs automatically. We find that templates can repair 5 mirror bugs. Repairing the remaining 4 bugs needs more complicated techniques, but mirror bugs provide hints for repairing these bugs.

2 EXAMPLE

To illustrate our target problem, this section introduces a sample mirror bug from `Hibernate`. `Hibernate` has both a Java implementation and a C# implementation, *i.e.*, `Hibernate` [21] and `NHibernate` [20]. It is a framework for programmers to simplify the interactions with databases, and it allows them to define the mapping relations between classes and database tables. For example, the below XML file defines a one-to-one mapping between `Person` and `Address`:

```xml
<class name="Person">
  <id name="Id" generator=" identity " />
  <one-to-one name="Address"/>
</class>
```

After the above mapping is defined, when a `Person` object is created, a corresponding `Address` object is created. To reduce the effort of copying data from the database, `Hibernate` implements a cache. If an object is fetched multiple times, `Hibernate` checks the cache to reduce the response time. Figure 1a shows a bug report of `Hibernate`. When a mapping is one-to-one, `Hibernate` does not check the cache to reduce the response time. As shown in Figure 1b, `NHibernate` has an identical bug. A programmer mentions that a workaround is to define the mapping as a many-to-one relation:

```xml
<class name="Person">
  <id name="Id" generator=" identity " />
  <many-to-one name="Address" column="Id" ... />
</class>
```

When mappings are one-to-one, the buggy versions of `Hibernate` and `NHibernate` do not check whether an object is in the cache. We inspected the patches for both the `Hibernate` and `NHibernate` bugs, and found some overlapped changes. For example, the changes in Figure 1c and Figure 1d are overlapped. The `assemble` and `Assemble` methods are called when one-to-one objects are created. In both patches, Line 2 does not check the cache, but the added lines fetch an object from the cache, if it is already created.

Many projects have implementations in different languages. As shown in our example, a bug in an implementation can be reproduced in the implementations in other languages. In our study, RQ1 explores the overall distributions. The bug in Figure 1 appears in both the Java and the C# implementations. We call such bugs two-sided bugs (an alternative term for mirror bugs). For two-side bugs, RQ2 explores how many such bugs are already fixed. The bug in Figure 1 is fixed on both sides. For unfixed two-side bugs, in Section 6.1, we build their patches according to their fixing commits, and report our new bugs and patches to issue trackers of the other implementations. For fixed two-side bugs, we conduct a pilot study to explore the potential and challenges of repairing them automatically in Section 6.2. If a bug appears in only the Java or the C# implementation, we call them one-side bugs. RQ3 explores why a bug from an implementation does not appear in the implementation of the other language.

3 METHODOLOGY

This section introduces our dataset (Section 3.1), and our analysis protocols (Section 3.2).

3.1 Dataset

In this study, we use the following criteria to select subject projects. First, the project must be implemented in Java and C#. We select this language pair, since the two languages are similar. Although mirror bugs exist in other programming language pairs, analyzing source files across two languages requires much more programming expertise. We leave the exploration of other language

Table 1. Dataset

Project	Language	Version	LOC	Developer	Star	Commit	Bug	Fixed		Open	
								Total	Sample	Total	Sample
Lucene [22]	Java	9.2.0	824,006	263	1,430	36,387	4,197	380	90	511	102
Lucene.NET [19]	C#	4.8.0	614,285	46	1,934	6,693	445	31	24	16	11
Hibernate [21]	Java	6.1	979,596	444	5,240	15,187	9,187	873	90	146	45
NHibernate [20]	C#	5.3.12	577,391	171	2,041	8,656	2,424	37	33	100	24
JTS [17]	Java	1.19.0	114,370	42	1,515	1,293	97	58	48	41	17
NTS [18]	C#	2.5	115,948	25	1,130	2,176	60	23	19	5	3
Log4j2 [23]	Java	2.18	175,183	142	1,990	12,462	1,834	367	90	368	30
Log4net [8]	C#	2.0.14	28,946	13	700	1,203	412	18	8	73	4
Total			3,429,725	1,146	15,980	84,057	18,656	1,787	402	1,260	236

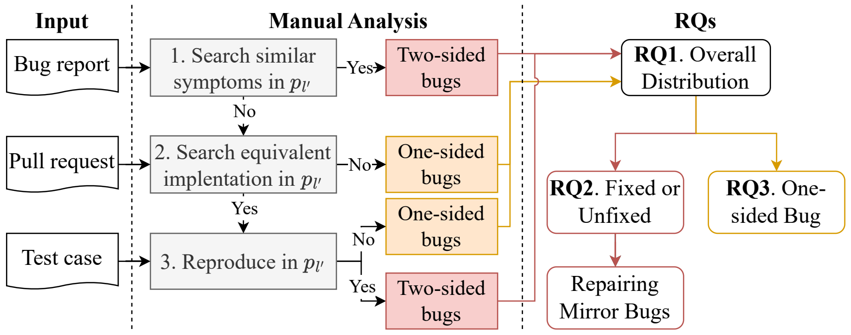


Fig. 2. The classification framework.

pairs to future work. Second, the project must have an issue-tracking system (*e.g.*, JIRA [24]) and a code repository. With the two systems, it is feasible to learn the details of bugs and their fixes. Finally, the project must have more than 300 issue reports, so we can collect sufficient bug reports for analysis. Following the three criteria, we select 4 projects, since they are well-known and widely used. Table 1 shows our selected projects. `Lucene` [22] and `Lucene.NET` [19] are text search engines. `Hibernate` [21] and `NHibernate` [20] are object and relational mapping frameworks that help programmers interact with databases. `JTS` [17] and `NTS` [18] are libraries for creating and manipulating vector geometry, and they are popularly used in geographic information systems. `Log4j2` [23] and `Log4net` [8] are logging frameworks that help record application behaviors. We selected the four projects from Github, since their C# implementations are all ported from Java implementations. For example, the website of `NHibernate` explains that it starts as a port of `Hibernate`. As a result, it is easier to locate their mirror bugs. In Column "Project", each row denotes an implementation in either Java or C#. Column "Language" lists the programming languages of implementations. Typically, projects in different languages have separate websites. For example, `Lucene` [22] and `Lucene.NET` [19] have two different websites. As our study involves much manual inspection effort, we cannot afford the effort of analyzing too many issue reports. Although there are many other cross-language projects, from the four projects, our collected subjects are competitive with those of prior empirical studies (*e.g.*, [98]).

Column "Version" lists the latest versions. Except `Lucene.NET` and `Lucene`, all projects have close versions of Java and C# implementations. Besides, all projects except `JTS` and `NTS` support priority flags for their bug reports. Column "LOC" lists the lines of code. `Lucene` and `Hibernate` are large projects. In total, the four projects have more than three million lines of code. Column "Developer" lists the number of developers. All the Java implementations have more developers than their C# correspondences. In Column "Star", the numbers show the number of stars on Github. Although the version `Lucene.NET` is much older than `Lucene`, it has more stars. In all the other projects, the

Java implementations have more stars than their C# implementations. Column "Commit" lists the number of commits. We notice that a higher version has more commits. Except for `JTS` and `NTS`, the Java implementations are more actively maintained than their C# implementations. For example, when we conducted this study, the current versions of `Lucene.NET` and `Lucene` were `4.8.0` and `9.2.0`, respectively. Although `Lucene.NET` is similar to `Lucene 4.8.0`, it becomes quite different from `Lucene 9.2.0`. As we deliberately select projects with diverse backgrounds, the four selected projects show different patterns in the above aspects.

Column "Bug" lists the number of all bug reports. For all the other projects, the Java implementations have much more bug reports than their C# implementations do. As `JTS` and `NTS` are niche libraries, both projects have fewer than 100 bug reports. In our study, we select both fixed and open bugs. We use two criteria to select fixed bugs: (1) bug reports must be created after 2017, and (2) bug reports must be blocker, critical, and major ones. The second criterion does not apply to `JTS` and `NTS` since they do not have priorities. Column "Fixed" lists the number of our selected fixed bugs. Besides the above two criteria, we use two other criteria to select open bugs: (1) bug reports must include test cases or pull requests, and (2) bug reports must be confirmed and not workarounds. Column "Open" lists the number of our selected open bugs. Subcolumn "Total" presents the total selected open bug reports. Subcolumn "Sample" presents the number of open bug reports selected for analysis. In total, we analyzed 236 samples.

Column "Fixed" lists the number of our selected fixed bugs. In total, as shown in subcolumn "Total", we have 1,787 candidate fixed bug reports. From these candidates, we select bug reports in the following order. The bug reports of `JTS` and `NTS` have no priorities. For the two projects, we select bug reports from the latest ones. For the other three projects, we select bug reports in order of their priorities. For each implementation, we terminate the selection, if all candidates are inspected or 90 bug reports are selected. Subcolumn "Sample" lists the number of our final selected fixed bug reports. In total, we have analyzed 402 samples.

Due to the heavy manual effort, it is infeasible to manually inspect so many bug reports. Researchers typically analyze fewer bug reports in their empirical studies. For example, Xuan *et al.* [98] analyze 300 bugs in their empirical study. Furthermore, in their studies, researchers do not align and analyze bugs in different languages. Although our analysis is more difficult, we already analyzed more bugs.

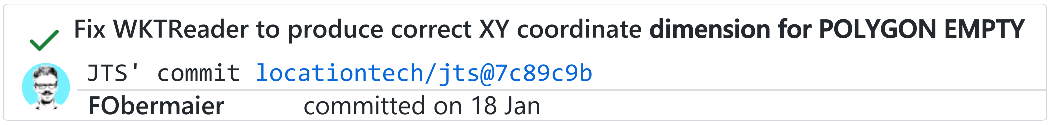
3.2 Analysis Overview

Figure 2 shows our classification framework. The input for classification comprises a bug report. The bug report can have a pull request or a test case. First, we search for bug reports with similar symptoms in the other-side projects. If no such reports are found, we then examine the available pull requests to identify equivalent implementations. If we have test cases, we translate these test cases and attempt to reproduce the symptoms to determine whether the bug is two-sided or one-sided. As introduced in Section 1, our study has three research questions. RQ1 explores the overall distributions. In this RQ, we classify bugs into two-sided and one-sided bugs. Two-sided bugs appear in both the Java and the C# implementations, but one-sided bugs appear in only the Java or the C# implementation. For two-sided bugs, RQ2 explores how many of them are already fixed. If they are unfixed, in Section 6.1, we try to write their patches according to their fixing commits. After that, we report our new bugs and pull requests to collect feedback from programmers. In addition, we explore the challenges of automating the repairing process in Section 6.2. For one-sided bugs, RQ3 explores why it is infeasible to trigger their symptoms in other languages.

Our manual inspection includes reading the introductions and manuals of the projects; inspecting bug reports to understand the symptoms; and inspecting patches to understand the causes. Our

```
//WKTReader.java
private Polygon readPolygonText(StreamTokenizer tokenizer ,
  EnumSet<Ordinate> ordinateFlags ) {  ...
  if (nextToken.equals(WKTConstants.EMPTY)) {
-   return geometryFactory.createPolygon() ;
+   return geometryFactory.createPolygon(createCoordinateSequenceEmpty (...) ) ;
 }...  }  ...
```

(a) The patch of JTS#827



✓ Fix WKTReader to produce correct XY coordinate **dimension for POLYGON EMPTY**

JTS' commit locationtech/jts@7c89c9b

FObermaier committed on 18 Jan

(b) A NTS commit[10] that mentions the JTS patch

```
//WKTReader.cs
private Polygon ReadPolygonText(TokenStream tokens,
  GeometryFactory factory, Ordinates ordinateFlags ) {  ...
    if (nextToken.Equals(WKTConstants.EMPTY)) {
-   return factory .CreatePolygon();
+   return factory .CreatePolygon(CreateCoordinateSequenceEmpty(...))
  }...}...
```

(c) The modifications of the NTS commit[10]

Fig. 3. A two-sided bug

classification results can be affected by human bias. To reduce the bias, the two authors independently classified the bugs according to the same protocols. We then apply Krippendorff's α test [70] to measure the inconsistency of our classification results. Krippendorff's α value is between zero and one, where zero denotes a random chance and one denotes a perfect agreement. Initially, the value for our classification is 0.9102, which indicates statistical confidence. To get the final results, we discuss the inconsistent classifications and attempt to achieve perfect agreement. If inconsistent results still exist, another author makes the final decision based on their discussions.

4 EMPIRICAL RESULT

This section introduces our empirical results. More details of our study are on our website: https://anonymous.4open.science/r/cross-language-DD47

4.1 RQ1. Overall Distribution

4.1.1 Protocol. To answer this research question, we classify bugs into one-sided bugs and two-sided bugs. For each bug report within one of the paired projects, p_l, we search the corresponding bug reports in the other project, $p_{l'}$, to identify a report with similar symptoms. This search is facilitated by reading and comparing the symptoms detailed in the two bug reports to determine their similarity. For example, after reading the bug report in Figure 1a, we determine that this bug is related to one-to-one mappings. We then search the bug reports of `NHibernate` with the keyword, "one-to-one". From the retrieved bug reports, we identify the bug report in Figure 1b. The two bug reports describe a similar symptom in fetching second-level caches. If we can find such a report, we examine the fixing commits to verify if they resolve the similar bug, thereby confirming the bug as two-sided.

If we cannot find a bug report with similar symptoms, we locate its buggy files in p_l according to its fixing commit. Subsequently, we search for equivalent source files in the latest version of $p_{l'}$ by identifying files with similar names. As ported implementations have many similar code

```
//TermsEnum.java
public abstract class TermsEnum implements BytesRefIterator { ...
 public static final TermsEnum EMPTY =
 -  new BaseTermsEnum() {
 +  new TermsEnum() {
 +     private AttributeSource atts = null;
          ...}  }
//BaseTermsEnum.java
public abstract class BaseTermsEnum extends TermsEnum {...}
```

(a) fixing patch for LUCENE-9661

```
//TermsEnum.cs
public abstract class TermsEnum : IBytesRefEnumerator{ ...
 public static readonly TermsEnum EMPTY = new TermsEnumAnonymousClass();
 private class TermsEnumAnonymousClass:TermsEnum
    {...}    ...}
```

(b) unchanged counterpart in Lucene.NET

Fig. 4. A one-sided bug

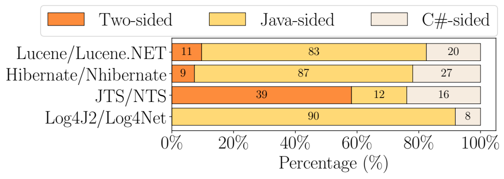
fragments, it is straightforward to locate many equivalent source files. For example, the source files in Figure 1c are quite similar with the code lines in Figure 1d. If we can reproduce a similar symptom on the equivalent source files of $p_{l'}$, we classify this bug report as a two-sided bug. If a bug is simple, it is straightforward to reproduce its symptom. If it is complicated, programmers often implement test cases to verify whether their patches work as expected. We translate these test cases to the other language to check whether similar symptoms can be triggered. If equivalent source files do not exist, we identify it as a one-sided bug.

For open bug reports, if a bug report is open and its pull request is accepted or merged in the latest version, we follow the above procedure to determine whether it is a two-sided bug or one-sided bug. In instances where a pull request is unavailable, we attempt to reproduce the bug symptoms in p_l using the provided test case. Subsequently, we translate these test cases into another language to ascertain whether similar symptoms can be triggered in $p_{l'}$. If both versions exhibit similar bugs, we classify the bug report as a two-sided bug; conversely, if we cannot reproduce the similar symptoms in $p_{l'}$, we categorize it as a one-sided bug.

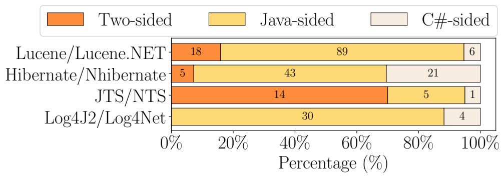
4.1.2 Result. We classify the bugs into two categories:

T1 Two-sided bugs (96/638, 15.0%). The bugs in this category appear in both Java and C# implementations. In particular, mirror bugs account for 14.7% of fixed bug reports (59/402) and 15.7% of open bug reports (18/236). The ratio of open bug reports is slightly higher than that of fixed bug reports. As for programming languages, the ratio of Java implementation is 18.8% (96/512), and the ratio of C# implementations is 76.2% (96/126). Programmers can actively locate and fix mirror bugs. For instance, `JTS` [17] is a Java library for creating and manipulating vector geometry. A bug report of `JTS` [25] complains that empty `Polygon` objects are created as 3-dimensional data, but they shall be two-dimensional data. Figure 3a shows its patch. The buggy code at Line 5 calls `createPolygon` without any arguments, and it sets the dimension to 3. To fix the bug, Line 6 passes an argument to `createPolygon`, and `Polygon` objects are created with the correct dimension. `NTS` is the C# implementation of `JTS`. As shown in Figure 3b, `NTS` programmers can read this bug, and modify their code accordingly. Figure 3c shows the modifications on `NTS`, and they are quite similar to the modifications on `JTS` as shown in Figure 3a.

T2 One-sided bugs (542/638, 85.0%). We determine that a bug report is one-sided, if we cannot reproduce its symptoms on the corresponding implementation using the other language. For



(a) Fixed bug reports.



(b) Open bug reports.

Fig. 5. The distribution of one-sided and two-sided bugs.

example, `Lucene-9661` [11] reports a random hang. As shown in Figure 4a, Line 4 of the `TermsEnum` class creates an instance of `BaseTermsEnum`. To create the instance of `BaseTermsEnum`, it needs to call `TermsEnum`, since as shown in Line 9 `BaseTermsEnum` is a subclass of `TermsEnum`. This process constructs a class initialization cycle. As Java Virtual Machine has a unique initialization lock [34], when it loads `TermsEnum` and `BaseTermsEnum` at two different threads simultaneously, a thread (t_a) can hold the lock for `TermsEnum` and another thread (t_b) can hold the lock for `BaseTermsEnum`. When this happens, it causes a deadlock. As shown in Figure 4a, to fix the bug, Line 5 replaces `BaseTermsEnum` with `TermsEnum`. `Lucene.NET` does not implement the corresponding class of `BaseTermsEnum`. As shown in Figure 4b, `EMPTY` is simply initialized by a private class `TermsEnumAnonymousClass`. Due to the different implementations, `Lucene-9661` does not appear in `Lucene.NET`. The above observations lead to a finding:

Finding 1. In total, 15.0% of sampled bugs appear in both the Java and C# implementations. C# implementations have a higher ratio of mirror bugs than Java implementations.

Figure 5 shows the distribution of different projects. The orange bars denote two-sided bugs; the yellow bars denote bugs that appear on only Java implementations; and the grey bars denote bugs that appear on only C# implementations. When `Log4j` upgrades from `1.x` to `2.x`, most source files are rewritten. The implementation of `Log4net` is similar to `Log4j 1.x`, and `Log4j2` becomes quite different from `Log4net`. As we select recent bugs, the bugs from `Log4net` are similar to `Log4j 1.x`, but we find no two-sided bugs between `Log4j2` and `Log4net`. In contrast, `JTS` and `NTS` have more two-sided bugs. In RQ2, we find that the programmers of `NTS` actively read and fix bugs that are reported to `JTS`. For example, as shown in Figure 3b, the message of the `nts@60eed6b` [10] commit in `NTS` has a link to the `jts@7c89c9b` commit in `JTS`. Please refer to Section 4.2 for a detailed analysis.

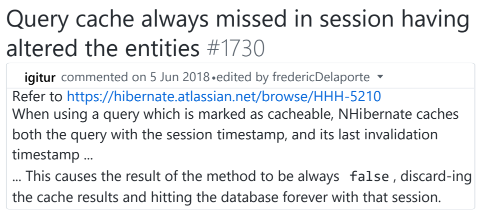
In summary, 15.0% of bugs are two-sided. Considering the two implementations may become quite different during their evolution, the percentage is attractive enough for further investigation. Also, there can be some opportunities in researching mirror bugs, since more mirror bugs can be fixed when programmers (*e.g.*, `JTS`) pay attention to such bugs.

4.2 RQ2. Fixed or Unfixed

4.2.1 Protocol. In this section, we analyze how many two-sided bugs were fixed. For some bug reports in p_l, we find their corresponding bug fixes of $p_{l'}$, and classify these bug reports into **T1.1**. Based on whether their bug reports of $p_{l'}$ are identified, we refine this category into **T1.1.1** and **T1.1.2**. We put unfixed two-sided bugs into **T1.2**, explore why they are unfixed, and refine this category based on the causes.

4.2.2 Result. We identified two types of two-sided bugs:

T1.1 Fixed two-sided bugs (63/96, 65.6%). In this category, a bug is already fixed in the Java implementation and the C# implementation. We further refine them according to the fixing process we observe.

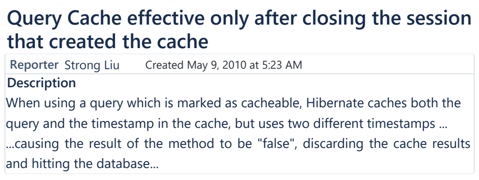


Query cache always missed in session having altered the entities #1730

igitur commented on 5 Jun 2018 •edited by fredericDelaporte

Refer to https://hibernate.atlassian.net/browse/HHH-5210
When using a query which is marked as cacheable, NHibernate caches both the query with the session timestamp, and its last invalidation timestamp ...
... This causes the result of the method to be always `false`, discard-ing the cache results and hitting the database forever with that session.

(a) Part of `nhiberante#1730` descriptions from `NHibernate`.



Query Cache effective only after closing the session that created the cache

Reporter Strong Liu Created May 9, 2010 at 5:23 AM

Description

When using a query which is marked as cacheable, Hibernate caches both the query and the timestamp in the cache, but uses two different timestamps ...
...causing the result of the method to be "false", discarding the cache results and hitting the database...

(b) Part of `HHH-5210` descriptions from `Hibernate`.

Fig. 6. An example of both-sided bug reports

T1.1.1 Fixed mirror bugs with both bug reports (15/96, 15.6%). In this category, a bug is reported to both the Java implementation and the C# implementation. The two bug reports are often similar. For example, as reported by `HHH-5210`, on the Java side, a timestamp shall be set to the time when a query result is cached, but it is set to the created time of the current session. In `NHibernate#1730`, `NHibernate` programmers confirm that the identical bug exists on the C# side. Figure 6 shows the two bug reports. The two reports are created by different developers, but `NHibernate#1730` has a reference to `HHH-5210`. `NHibernate#1730` even copies many sentences from `HHH-5210` to describe the bug.

As the developers of open-source projects often change, even if we can find a bug report, it is difficult to determine whether the reporter is a user or a developer. As a user is unlikely to use both the Java and the C# implementations, the bug reports can be filed by developers.

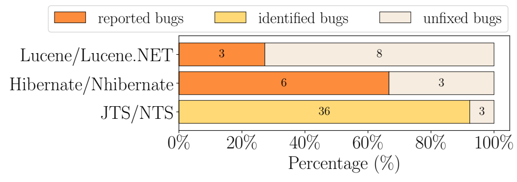
T1.1.2 Fixed mirror bugs with one-side bug reports (48/96, 50.0%). In this subcategory, a bug is reported to only one side, but programmers still fix the same buggy symptoms in both the Java and C# implementations. Also, we find evidence indicating they may actively learn the patches for the same bugs in the other implementation. For example, as shown in Figure 3, after a programmer fixed `JTS#827` [25], another developer fixed the same bug symptom on `NTS`. We believe that `NTS` programmers learn how `JTS#827` is fixed, since the message of the `NTS` fixing commit has a link to the `JTS` commit. The above observations lead to a finding:

Finding 2. In total, 63 (65.6%) of our found 96 two-sided bugs are already fixed. Among them, 15 have bug reports in both Java and C# implementations, but 48 bugs have bug reports only on one side.

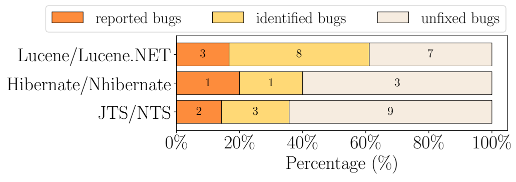
T1.2 Unfixed two-side bugs (33/96, 34.3%). In this category, we can find buggy methods on both sides. We further refine this category based on whether fixes have been implemented.

T1.2.1 Potential confirmed bugs (10/96, 10.4%). In this subcategory, we can find buggy methods on both sides, and we find fixes on only one side. We have produced the symptoms of these bugs in the other implementations, and write their patches. Among these unfixed mirror bugs, more than half of them are real bugs, but remain unfixed by developers. For example, `NTS#567` [12] complains that `GeometryFixer` changes the dimension of the coordinates when the input is a polygon. As shown in Figure 9a, the dimension change happens at Line 3, when it sets the dimension for `z`. This dimension is invalid when `z` is not a number. The fixed code checks `z` before the assignment. As shown in Figure 9b, the Java counterpart has an identical bug. We further analyze these bugs in Section 6.1.

T1.2.2 Potential unconfirmed bugs (17/96, 17.7%). In this subcategory, we can find buggy methods on both sides, but neither side fixes them. We find potential unconfirmed bugs only from open bug reports. For instance, `LUCENENET#964` [38] complains that the `GroupingSearch` class may

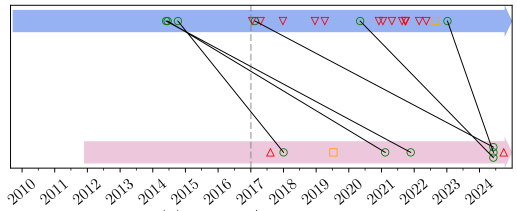


(a) Fixed bug reports.

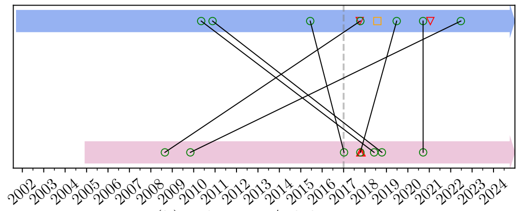


(b) Open bug reports.

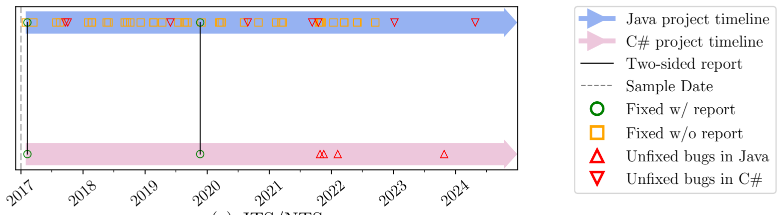
Fig. 7. The distribution of two-sided bugs.



(a) Lucene/Lucene.NET



(b) Hibernate/Nhibernate



(c) JTS/NTS

Fig. 8. The lifetime of two-sided bugs.

miss specific groups, even if such groups are defined in `IndexSearcher.Search`. The reporter of this bug provided a test case [39] to reproduce the symptom. As this bug report is still open, we reproduce this bug on the latest `Lucene.NET`. Furthermore, we translate this test case to the Java, and observe the same symptoms in the latest `Lucene`.

T1.2.3 Minor symptoms (6/96, 6.2%). These bugs have minor symptoms in p_l, and programmers of $p_{l'}$ may not fix them. For example, the symptom of Lucene-10118 [13] is that the log messages in `ConcurrentMergeScheduler` are too simple. The programmers of `Lucene` may not pay attention to this minor bug. The above observations lead to the following finding:

Finding 3. We have produced the symptoms of 81.8% (27/33) of unfixed two-sided bugs.

Figure 7 shows the distribution of different projects. `Log4j2` and `Log4net` are not included, since we found no two-sided bugs in the two implementations. Table 1 shows that `Lucene` and `Hibernate` have many bug reports. Their programmers can have too many bugs to repair, so they do not actively fix mirror bugs. As a result, we found that all mirror bugs in the two projects have bug reports. In contrast, `JTS` and `NTS` have fewer than 100 bug reports, and we find that no mirror bug in this project has bug reports. Meanwhile, as programmers do not have many bugs to repair, they actively learn the bug reports from the other implementations. As a result, we find many identified mirror bugs in the two projects. Meanwhile, `Lucene` and `Hibernate` do not actively fix mirror bugs. As a result, we find more unfixed mirror bugs in the two projects. Although the programmers of `JTS` and `JTS` fixed many mirror bugs, we still found unfixed ones.

Figure 8 shows the lifespan of two-sided bugs. The light blue and light red arrows represent the timelines of Java and C# projects, respectively. The gray dotted line indicates the start time of sampling bug reports. The green circles depict fixed mirror bugs with both bug reports (T1.1.1). Their positions are their reported dates. The black lines link Java and C# reports of mirror bugs. The three projects have different temporal patterns. For Lucene and Lucene.NET, Java bugs are reported earlier than their C# counterparts. In Hibernate and NHibernate, both Java and C# bugs can be firstly found in the other sides. For JTS and NTS, the two mirror bugs are reported at the same time. The orange squares represent fixed mirror bugs with one-side reports (T1.1.2). Their positions are the reported dates of one-side bug reports. The red triangles denote unfixed two-sided bugs. Their positions are the reported dates of known bug reports from the other sides. The orientation of each triangle indicates in which project bugs remain unfixed. In particular, an upward triangle (△) denotes that a bug is unresolved in a Java project, and a downward triangle (▽) indicates that a bug remains unfixed in a C# project. For instance, in Figure 8 (a), a red downward triangle in the blue arrow denotes that we can find the mirror bug of a Lucene bug, but this mirror bug is unfixed in the latest version of Lucene.Net. The three projects follow different patterns. The development of Lucene.NET significantly lags behind that of Lucene, and it does not support many features of the latest Lucene. As a result, Lucene provides more mirror bugs than Lucene.NET. Both Hibernate and NHibernate are under active maintenance and have many bug reports. As a result, Hibernate and NHibernate both provide mirror bugs and are mutually beneficial. NTS is not as popular as JTS. As it has only a few bug reports, the developers of NTS actively identify and repair mirror bugs from JTS. The above observations lead to a finding:

Finding 4. Detecting mirror bugs is beneficial when the other sides have many bug reports.

In summary, we find that users seldom report bugs to multiple implementations, since they typically use projects in a language. However, according to the results from JTS and NTS, many mirror bugs can be identified, if their programmers actively identify and fix such bugs. Although it is difficult to set up the programming contexts (*e.g.*, databases), we have produced the symptoms of 27 unfixed two-sided bugs.

4.3 RQ3. One-sided Bug

4.3.1 Protocol. In Section 4.1, we classify bugs into two categories. The bugs in the **T2** category are found in only one implementation, and cannot be reproduced in the other. In this research question, we analyze why these bugs appear in only an implementation of a project. Our analysis has the following steps. First, we read each bug report to learn its symptoms. Second, we investigate its fixing commits to learn its modified locations and causes. Third, we search the source files of the other-language implementation for correspondences. If we cannot find such locations, we classify them into subcategories of **T2.1**, according to the granularity of missing correspondences. If we find the correspondences of a bug, we further analyze why it is infeasible to produce the symptoms. We then classify this bug into subcategories of **T2.2** and or **T2.3**, according to its reason.

4.3.2 Result. Our identified categories are as follows:

T2.1 No corresponding source files (385/542, 71.0%). We failed to find their corresponding buggy locations.

T2.1.1 File differences (190/542, 35.0%). In this subcategory, we can find corresponding modules in p_l and $p_{l'}$, but their files are different. For example, LUCENE-10401 [26] reports that the codec in Lucene lacks an empty check. The codec is a component of Lucene, and it is responsible for reading and writing index files. The buggy codec mentioned by LUCENE-10401 is Lucene90, but Lucene.NET uses an old codec whose version is Lucene46. As the two implementations of codec have many different files, we cannot reproduce the symptom of LUCENE-10401 on Lucene.NET.

```
// RobustLineIntersector.cs ...
private ... CopyWithZ(Coordinate p, double z) {
-   var pCopy = new CoordinateZ(p);
-   if (!double.IsNaN(z)) pCopy.Z = z;
-   return pCopy;
+   Coordinate res;
+   if (double.IsNaN(z)) res = p.Copy();
+   else res = new CoordinateZ(p) { Z = z };
+   return res;
 }...
```

(a) The patch of NTS#567 [12].

```
// RobustLineIntersector.java ...
private ... copyWithZ(Coordinate p, double z) {
 Coordinate pCopy = new Coordinate(p);
 if (!double.IsNaN(z)) pCopy.setZ(z);
  return pCopy; }...
```

(b) The JTS code equivalent with the buggy code of NTS#567

Fig. 9. An example of potential bugs

T2.1.2 In-file differences (93/542, 17.2%). In this subcategory, buggy files in p_l have their corresponding files $p_{l'}$, but the implementation details are different. For example, as we introduced in Section 4.1.2, LUCENE-9661 reports a deadlock in LUCENE. As shown in Figure 4, although the buggy files are found in both the Java and C# implementations, they lock the resource differently. As a result, the deadlock does not appear in the Lucene.NET.

T2.1.3 Module differences (102/542, 18.8%). In this subcategory, we cannot find the corresponding modules. For example, LUCENE-10260 [14] reports a display problem with luke, a visualization module of Lucene. As Lucene.NET has no visualization module, we cannot reproduce this bug.

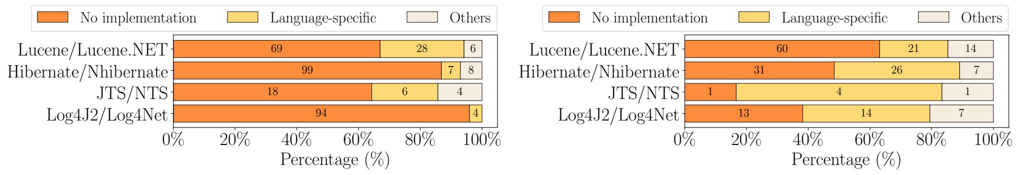
Finding 5. In total, 71.0% of one-sided bugs are caused by implementation differences.

T2.2 Language-specific bugs (110/542, 20.3%). In this category, bugs are caused by languages and their development environments.

T2.2.1 Build bugs (35/542, 6.5%). As programming languages have their own building tools, build bugs are seldom across languages. For example, Lucene-10042 [15] reports that the minimum JDK version is set at the wrong place in gradle [27] scripts. As gradle is a Java building tool, this bug will not affect Lucene.NET.

T2.2.2 Languages and frameworks (58/542, 10.7%). The bugs in this subcategory are caused by language-specific features and frameworks. For example, Lucene code calls a Java API, System.nanoTime(). As C# does not have the corresponding API, Lucene.Net calls DateTime.UtcNow.Ticks * 100 to mimic System.nanoTime(). The calculation is wrong since a unit tick is equivalent to 100 nanoseconds. As a result, a LUCENENET bug report [16] complains that a method hangs when it is executed. To fix this problem, the above call is replaced by J2N.Time.nanoTime(), which is declared by a third-party library, J2N.

T2.2.3 Library-related bugs (17/542, 3.1%). In this category, bugs are caused by dependencies. For example, Lucene-8175 [2] is caused by a concurrency bug in ICU4J. ICU4J is an API library providing Unicode services. Lucene.NET calls ICU4N, a C# implementation of ICU4J, but it does not have the concurrency bug.



(a) Fixed bug reports. (b) Open bug reports.

Fig. 10. The distribution of one-sided bugs.

> **Finding 6 .** In total, 20.3% of one-sided bugs are caused by languages and development environments.

T2.3 Others (47/542, 8.7%). In this category, we cannot reproduce bugs, due to various minor reasons.

T2.3.1 Hard to trigger (22/542, 4.1%). These bugs are reported to p_l, but we fail to write test cases to trigger their corresponding symptoms from $p_{l'}$. For example, the test case of `Lucene-10598` [28] calls the `SortedSetDocValues.docValueCount` method to calculate the unique ordinals of an indexed document. In the latest `C#` implementation, this method is unimplemented. As a result, it is difficult to reproduce this bug on `Lucene.NET`.

T2.3.2 Migration bugs (11/542, 2.0%). Some bugs are introduced during the migration from p_l to $p_{l'}$, and such bugs do not appear in p_l. An example is `NTS#380` [9]. As shown below, Line 5 calls a wrong method. A programmer mentions that `JTS` calls the right method, ever since the oldest traceable commit. As `NTS` is migrated from `JTS`, `NTS#380` shall be introduced during the migration process.

```
1  //GeometryTransformer.cs
2  public class GeometryTransformer{...
3   public ... Transform(Geometry inputGeom){...
4    if (inputGeom is LinearRing)
5  -  return TransformLineString((LinearRing)inputGeom, null);
6  +  return TransformLinearRing((LinearRing)inputGeom, null) ;... } ... }
7  //GeometryTransformer.java
8  public class GeometryTransformer{...
9   public ... Transform(Geometry inputGeom){...
10   if (inputGeom instanceof LinearRing)
11    return TransformLineString((LinearRing)inputGeom, null) ;... } ... }
```

T2.3.3 Reverted changes (14/542, 2.6%). In this category, the modifications of fixing a bug are reverted, before these modified files are translated to the other project. For example, when `Lucene` programmers implement multiple threads to delete documents, they delete Lines 7 to 9 as shown in the below code fragments:

```
1  //DocumentsWriter.java
2  final class DocumentsWriter{
3    private final IndexWriter writer; ...
4    private boolean doFlush (...) ... {
5      boolean hasEvents = false;
6      while(flushingDWPT != null) {...}
7  -   if (hasEvents) {
8  -     writer.doAfterSegmentFlushed(false, false);
9  -   } ...
10   } ... }
11 //DoucmentsWriter.cs
```

```
12 internal sealed class DocumentsWriter {...
13   private bool DoFlush (...) {
14     bool hasEvents = false;
15     while (flushingDWPT != null) {...}
16     if (hasEvents) {
17       PutEvent(MergePendingEvent.INSTANCE);
18     } ...
19   } ... }
```

As reported in LUCENE-7894 [1], a thread wrongly counts the number of documents. To fix the bug, the deletions of Lines 7 to 9 are reverted. Lucene.NET does not implement multiple threads support for deleting documents, and the corresponding Lines 16 to 18 are never deleted. As a result, LUCENE-7894 [1] cannot be reproduced on Lucene.NET.

Figure 10 shows the distributions of different projects. In total, we cannot reproduce 71.0% of bugs, since we cannot locate their correspondences in the other implementations. We search for similar source files, but their correspondences can be dissimilar. As we are outsiders to these projects, we may not find some hidden correspondences, and thus overestimate the percentage of T2.2. If programmers replicate our study, they can find more mirror bugs.

In summary, in the majority of one-sided bugs, we fail to find corresponding implementations (71.0%). The others cannot become mirror bugs because they are language-specific (20.3%) or because of other minor reasons (8.7%). Here, if their programmers replicate our study, the percentage for no corresponding implementation could be lower, since they can locate more correspondences of buggy source files.

5 THE SIGNIFICANCE OF OUR FINDING

In this section, we interpret the significance of our findings.

Understanding the importance of mirror bugs. Although we can ignore valid cases, Finding 1 shows that 15.0% of all inspected bugs appear on both sides. As shown in Finding 2, only 23.8% of fixed two-sided bugs are explicitly reported. Programmers can actively find and repair mirror bugs. For example, we find that several programmers contribute to both JTS and NTS, and they actively fix mirror bugs. As a result, in this project, we found more mirror bugs. If suitable programmers are invited or proper tools are proposed, it is feasible to detect mirror bugs from more projects. Log4net is ported from log4j, but we sampled bugs from log4j2. As a result, we find no mirror bugs for this project. Still, when Log4net is updated to log4j2 in the future, it should be feasible to detect mirror bugs from this project. Although we do not categorize mirror bugs based on their symptoms, we do categorize them according to the locations where they occur. It would be worthwhile to conduct a new empirical study to explore the categorization of mirror bugs' symptoms.

Detecting and repairing mirror bugs. Like JTS and NTS, an application can have imbalanced users, and a version can attract many more users than other versions. The bug reports of the popular version can be useful for other versions. Although we manually inspected only limited bugs, we have found and repaired 9 new mirror bugs. A tool can detect many more bugs if properly developed. Given a list of bugs, the tool must first filter out one-side bugs. We find that one-side bugs are caused by implementation differences (71.0%, Finding 5), language-specific issues (20.3%, Finding 6), or various minor reasons (8.7%). Researchers have investigated how to detect and repair cross-platform bugs [45, 72, 75, 110]. Motivated by these approaches, in Section 6, we try to repair manually unfixed mirror bugs and analyze the challenges of automating the repairing process.

Exploring evolution patterns of mirror bugs. In software, clones are common [87], and the evolution of clones is intensively studied [47, 67, 83]. These studies analyze clones in the same language. If we extend the definitions of clones, we can find clones across languages [56], and

Table 2. Our reported new mirror bugs

Original Id	Symptom	Priority	Our Id	Status
LUCENE-8755	`QuadPrefixTree` crashes when indexing at high precisions	Critical	LUCENENET#644	PR rejected
LUCENE-9940	`DisjunctionMaxQuery.equals()` is wrongly implemented	Major	LUCENENET#779	Open
LUCENE-10008	`CommonGramsFilterFactory` ignores the `ignoreCase` setting	Major	LUCENENET#780	PR accepted
LUCENE-10059	`JapaneseTokenizer` throws `AssertError` with backtrace	Major	LUCENENET#775	PR accepted
LUCENE-10441	`ArrayIndexOutOfBoundsException` in during indexing	Major	LUCENENET#1003	PR accepted
LUCENE-8614	`ArrayIndexOutOfBoundsException` in `ByteBlockPool`	Major		
HHH-14413	`EntityUpdateAction` increments the version when updating is vetoed	Blocker	NH#3198	PR accepted
NH#1419	`IsDirty` throws exceptions for transient many-to-one objects	Major	HHH-15848	Open
NTS#567	`GeometryFixer` does not keep the dimensions of target coordinates	N/A	JTS#919	PR accepted
NTS#589	`WKTReader` creates 3-D coordinates when reading 2-D ones	N/A	JTS#939	Open

mirror bugs are cross-language clones with bugs. On one hand, the known knowledge of clones is useful for understanding mirror bugs. For example, Kim *et al.* [67] show that after programmers copy a code fragment from a code location, this code fragment typically becomes dissimilar during evolution. We notice that when an application is initially ported from Java to C#, its C# source files are quite similar to its Java source files, but they also become less similar with evolution. For example, As shown in Table 1, the latest versions of the Java implementation and the C# implementation are typically different, and their latest source files become quite different. On the other side, the evaluation of cross-language projects and mirror bugs can deepen the knowledge of the evolution of clone bugs. Unlike classical clones, cross-language projects can become similar, if a lagging version catches up. As a result, bug fixes can be useful for future versions, even if their correspondences are not implemented.

6 REPAIRING MIRROR BUG

As we summarize in Section 5, it is feasible to detect and repair new mirror bugs by learning known bugs. In this section, we try to fulfill this vision on our dataset by answering the following two research questions?

- **RQ4: Can we repair new mirror bugs by manually learning known bugs?**
- **RQ5: What are the challenges if researchers automate the process?**

6.1 RQ4. Repairing New Mirror Bugs

As we summarize in Section 5, it is feasible to detect and repair new mirror bugs by learning known bugs. In this section, we try to manually fulfill this vision on our dataset.

6.1.1 Protocol. As we introduced in Section 4.2.2 (T1.2.1), we identified 10 potential confirmed mirror bugs. To explore whether known bugs are useful for repairing mirror bugs, we try to repair these potential mirror bugs with their corresponding known bugs. Preparing patches necessitates a profound understanding and expertise in both Java and C# project implementations. Our steps are as follows. First, based on the descriptions of a bug report in p_l, we manually write test cases to reproduce the buggy symptoms on $p_{l'}$. Second, if we reproduce the symptom, we proceed to address it in $p_{l'}$ by referencing the corresponding fixing commit from p_l. We then manually translate the patch into the alternative language implementation. Third, if we fix the bug in $p_{l'}$, we report this bug and our fixed code to $p_{l'}$. Finally, we collect and analyze the feedback from developers.

6.1.2 Feedback from programmers. It takes much programming experience to identify and repair bugs from real projects. As outsiders, for our projects, we cannot even determine whether their source files have bugs or not. Fortunately, the known bugs in a language provide valuable references to identify the bugs in the other language. Based on the known bugs, we successfully located all the new buggy locations of the known 10 bugs. Based on their known patches, we fixed them according to the existing patches in the other languages. The 10 bugs are already confirmed and fixed on one side. After we reproduce them in the other implementations, all our found new bugs shall be true bugs. Still, we need feedback from programmers to understand their attitudes toward mirror bugs. As a result, we report our new bugs and their pull requests to the corresponding implementations. Table 2 shows the results. The first three columns list the IDs, the symptoms, and the priorities of the original bug reports. Columns "Our Id" lists the IDs of our bug reports. In Rows 6 and 7 of Table 2, `LUCENENET#1003` is a mirror bug of `LUCENE-10441` and `LUCENE-8614`. The two `Lucene` bug reports complain the same symptoms, and their pull requests are similar. Although this problem has been reported twice, both bug reports are still open. We find that `Lucene.Net` has the same bug and throws a corresponding exception. Column "Status" lists the status of our pull requests. Programmers have responded to 5 of our pull requests:

1. Five pull requests were already accepted. For example, as we introduced in **T2.1.2**, `NTS#567` [12] reports that its `GeometryFixer` class unintentionally changes the coordinate dimensions. In particular, given a two-dimensional coordinate as the input of its `fix` method, it constructs a 3-dimensional coordinate, and the additional dimension is assigned to `NaN`, which means "not a number". We find that the latest version of `JTS` also has a `GeometryFixer` class, and it has the same buggy symptom. Based on the patch of `NTS#567` as shown in Figure 9a, we prepare the following patch for `JTS`:

```
// RobustLineIntersector .java
private static Coordinate copyWithZ(Coordinate p, double z) {
−  Coordinate pCopy = new Coordinate(p);
−  if (!Double.isNaN(z)) pCopy.setZ(z);
−     return pCopy;
+  Coordinate res;
+  if (Double.isNaN(z)) res = p.copy();
+  else res = new Coordinate(p);
+  p.setZ(z);
+  return res;
 }...
```

The above patch does not fully resolve the bug, due to the implementation differences. However, the knowledge learned from Figure 9a helps us resolve the problem in other code locations. For example, by replacing the constructor with `copy()`, we fixed the identical problem in the `SegmentNode` class of `JTS`:

```
public class SegmentNode{
  public final Coordinate coord; // the point of intersection
  public SegmentNode(Coordinate coord, ...) {...
−  this.coord = new Coordinate(coord);
+  this.coord = coord.copy();
     ...}...}
```

We reported our found bug and its pull request (`JTS#919` [29]) to `JTS`. Our bug report was confirmed in three days, and our pull request was accepted two days after we submitted it.

2. A pull request was rejected but programmers confirmed that this pull request could be useful in future versions. In `Lucene`, the `QuadPrefixTree` class utilizes Quadtrees to index coordinates. Given a two-dimension point in a rectangle region, `QuadPrefixTree` repeatedly divides the current region into four quadrants, and enters the quadrant where this point is, until the region is smaller than a pre-defined proportion. `Lucene-8755` [3] complains that the `getCell` method of this class throws an `IndexOutOfBoundsException`, if a coordinate falls on the borders of a bounding box. To fix this bug, programmers write the following patch [30]:

```
//QuadPrefixTree.java
 public Cell getCell(Point p, int level) {...
-  build (...) ;
-  return cells.get(0);  ...
+  for (int lvl = 0; lvl < levelLimit; lvl++){
+   int c = battenberg(currentXmid, currentYmid, xp, yp);  ...
+   str.bytes[str.length++] = (byte)('A' + c);
+  }
+  return new QuadCell(str, rel);
}
```

The new implementation of `getCell` method properly handles the coordinates on borders. Although the old version is considered a bug, the programmers of `Lucene` decided to keep the buggy behavior. In another patch [31], programmers use a `Boolean` value, `robust`, to check whether the version is newer than `8.3.0`:

```
1   //QuadPrefixTree.java
2   + protected boolean robust = true;
3    protected SpatialPrefixTree newSPT() {
4   -  return new QuadPrefixTree(ctx,
5   +  QuadPrefixTree tree = new QuadPrefixTree(ctx,
6         maxLevels != null ? maxLevels : MAX_LEVELS_POSSIBLE);
7   +  tree.robust = getVersion().onOrAfter(Version.LUCENE_8_3_0);
8   +  return tree;
9   }
10   public Cell getCell(Point p, int level) {...
11  -  build (...) ;
12  -  return cells.get(0);
13  +  if (!robust) {...
14  +    buildNotRobustly (...) ;
15  +    if (!cells.isEmpty()) {return cells.get(0);}
16  +   }...
17  }
```

If the version is older than `8.3.0`, Line 7 sets `robust` to `false`. When `robust` is `false`, Line 15 returns the value as the buggy version does, *i.e.*, what Line 12 returns.

On the C# side, we find that `QuadPrefixTree` is a line-by-line translation of the Java implementation. Given the identical input, `Lucene.NET` throws an `ArgumentOutOfRangeException`. Based on the patch in Java, we prepared a patch for `Lucene.NET` to resolve the symptom. We report the bug [32] and submit our pull request [33]. However, the programmers of `Lucene.NET` rejected our pull request. It turns out that the latest version of `Lucene.NET` is still the equivalent of `Lucene 4.8.0`. `Lucene-8755` [3] fixes a bug in `Lucene 8.3.0`. The programmers of `Lucene.NET` believe that it is not the right time to merge our pull request, and they want to keep the buggy behaviors for backward compatibility. Still, they appreciate our report and pull request, admitting they are good references for their future versions. The feedback from programmers confirms our vision. The bug reports and patches from an implementation are useful to detect and fix bugs in the implementation in other languages.

In summary, according to the bugs in one-sided implementations, we found new mirror bugs in all three projects where two-sided bugs are found. In total, we have detected and fixed 9 such bugs. Even if the programmers of `NTS` actively learn and fix mirror bugs, we found two new bugs, and one of them has been accepted by their programmers.

Table 3. The successful cases of our manual study

C# Id	Java Id	Original	Naming	Keyword	API replacement
LUCENENET#559	LUCENE-5716	✗	✓	-	-
LUCENENET-598	LUCENE-6001	✗	✗	✗	✓
NH#1244	HHH-13456	✗	✗	✗	✓
NH-2011	HHH-15359	✗	✗	✗	✓
NH#1730	HHH-5210	✗	✗	✗	✓

6.2 RQ5. Challenge of Automation

As we summarize in Section 5, it is feasible to apply automated program repair techniques to automate the process of repairing mirror bugs. For instance, Liu *et al.* [74] propose template-based techniques to repair bugs. In this section, we conduct a preliminary study to explore the challenges of automation.

6.2.1 Protocol. Liu *et al.* [74] propose a template-based approach called TBar. We analyze this tool since it is a recent approach and mirror bugs provide a feasible way to construct templates. This tool does not include a technique to mine templates from C# patches. Even if we implement such a technique for C# code, templates from C# patches are unlikely to repair bugs in Java code. It takes too much effort to implement a tool that can extract templates from C# patches and translate them to templates in Java. Although we cannot provide the results of an automated tool in this RQ, we conduct a small-scale manual study to inspect the challenges of implementing such a tool.

In Section 4.2.2, we identified 9 mirror bugs with fixed bug reports on both sides. In this RQ, we analyze all the 9 mirror bugs. Following the idea of Liu *et al.* [74], we manually construct templates from their C# patches. After that, we use TBar to repair the corresponding Java bugs with original templates. If they fail to repair a Java bug, we sequentially apply the following modifications to the templates and use the revised templates to repair bugs:

1. Naming convention: We replace the names of method calls from the Pascal cases in C# to the camel cases in Java.

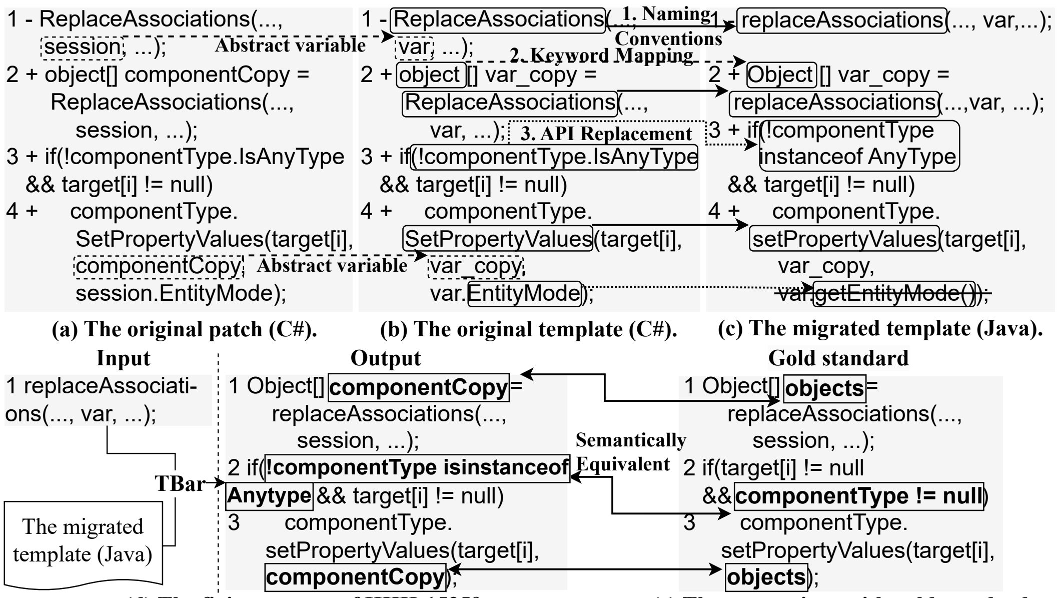
2. Keyword mapping: We replace C# keywords with corresponding Java keywords.

3. API replacement: We replace C# APIs with the corresponding Java APIs. This type of modification is quite challenging. Although researchers [52, 79, 108] have proposed various approaches to mine API mappings across languages, these approaches typically consider only the mappings of API methods. Besides replacing called API methods, we consider the replacements of their parameters, *e.g.*, swapping parameter orders and deleting unnecessary parameters.

We continue this process until a modified template works.

6.2.2 Learned Lesson. Our study leads to the following results:

1. **Modified templates can repair five mirror bugs.** Table 3 shows the five mirror bugs. Column "C# Id" and "Java Id" list the issue numbers of Java bug reports and C# bug reports, respectively. Column "Original" lists the result of original templates. Columns "Naming", "Keyword", and "API" list the results after we apply the corresponding modifications. For instance, `NH-2011` [40] reports that many-to-many relationships within a component are not correctly persisted when using `SaveOrUpdateCopy` or `Merge` in `NHibernate`. In particular, its `TypeHelper` class mishandles components during associations. As shown in Figure 11a, the buggy code calls `ReplaceAssociations` to replace the associations of the original component with the target component. It is buggy since it does not properly assign the transformed component to the target object. As shown in Figure 11a, the fixed code correctly sets the transformed component values and recognizes the many-to-many associations inside components. Based on this patch, we extract a template as depicted in Figure 11b. The dashed squares in Figure 11a and Figure 11b show the links between the patch and the template. This template fails to repair the mirror bug in Java. We applied all the three modifications, and Figure 11c shows our modified template. The links between Figure 11b and Figure 11c show



```
1 - ReplaceAssociations(...,
   session, ...);
2 + object[] componentCopy =
      ReplaceAssociations(...,
         session, ...);
3 + if(!componentType.IsAnyType
  && target[i] != null)
4 +     componentType.
      SetPropertyValues(target[i],
      componentCopy,
      session.EntityMode);
```

(a) The original patch (C#).

```
1 - ReplaceAssociations(...,
   var, ...);
2 + object[] var_copy =
   ReplaceAssociations(...,
      var, ...);
3 + if(!componentType.IsAnyType
  && target[i] != null)
4 +    componentType.
   SetPropertyValues(target[i],
      var_copy,
      var.EntityMode);
```

(b) The original template (C#).

```
1 - replaceAssociations(..., var,...);
2 + Object[] var_copy =
  replaceAssociations(...,var, ...);
3 + if(!componentType
    instanceof AnyType)
  && target[i] != null)
4 +    componentType.
  setPropertyValues(target[i],
     var_copy,
```
~~var.getEntityMode());~~

(c) The migrated template (Java).

```
1 replaceAssociati-
ons(..., var, ...);
```

```
1 Object[] componentCopy=
   replaceAssociations(...,
      session, ...);
2 if(!componentType isinstanceof
Anytype && target[i] != null)
3    componentType.
   setPropertyValues(target[i],
   componentCopy);
```

(d) The fixing process of HHH-15359.

```
1 Object[] objects=
   replaceAssociations(...,
      session, ...);
2 if(target[i] != null
  &&componentType != null)
3  componentType.
  setPropertyValues(target[i],
  objects);
```

(e) The comparison with gold standard.

Fig. 11. A successful template.

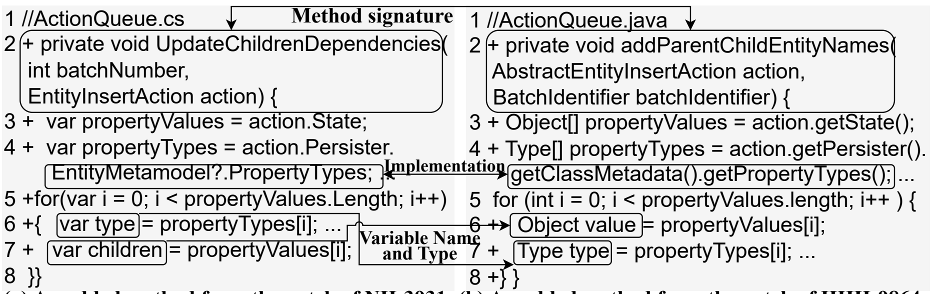
the modifications of naming conventions, keyword mappings, and API replacements in different styles of lines. For instance, `isAnyType` is modified to `isinstanceof AnyType`. In addition, although `Hibernate` has a corresponding method (`getEntityMode()`) for the `EntityMode` field in `NHibernate`, the `setPropertyValues()` method in `Hibernate` has only two parameters. As the last parameter of `SetPropertyValues()` is no longer necessary, we remove it from the template. Figure 11d shows the Java code after applying our modified template. Figure 11e shows the comparison with the gold standard. Although the code is different, it closely resembles the one written by programmers [41]. A notable difference between the two patches lies in Line 3. The generated patch checks whether `componentType` is an instance of `AnyType`, but the gold standard checks whether `componentType` is null. Our patch is stricter since the `instanceof` operator in Java will check whether an object is null before checking its type.

2. We failed to extract templates from the other four patches since they have hunks with only additions. For instance, NH-3931 [42] reports that inserting multiple parent entities leads to an incorrect order and a foreign key exception is thrown. Its patch has the hunk as shown in Figure 12a. As this hunk has no modified source, we cannot build the templates as described by Liu *et al.* [74]. Figure 12b shows the corresponding hunk in Java. Most code segments in the modified C# and Java code are similar. Although it is infeasible to define templates to fix this bug, we envisage that translating the fixed code and applying the translated code to the proper code location can fix the bug in Java. During the translation, many approaches such as API mapping [108], learning-based methods [81], and large language models [46] can be useful.

In summary, we confirm that templates can repair five mirror bugs, but they cannot fix the remaining four bugs. Still, the original templates cannot repair mirror bugs, and need modifications before they can work. Although templates cannot repair the remaining four bugs, mirror bugs provide details that are useful to repair bugs through other technical paths.

7 THREAT TO THE VALIDITY

Our study has the following threats.



```
//ActionQueue.cs
+ private void UpdateChildrenDependencies(
int batchNumber,
EntityInsertAction action) {
+  var propertyValues = action.State;
+  var propertyTypes = action.Persister.
   EntityMetamodel?.PropertyTypes;
+for(var i = 0; i < propertyValues.Length; i++)
+{  var type = propertyTypes[i]; ...
+  var children = propertyValues[i];
}}
```

```
//ActionQueue.java
+ private void addParentChildEntityNames(
AbstractEntityInsertAction action,
BatchIdentifier batchIdentifier) {
+ Object[] propertyValues = action.getState();
+ Type[] propertyTypes = action.getPersister().
getClassMetadata().getPropertyTypes(); ...
 for (int i = 0; i < propertyValues.length; i++ ) {
+ Object value = propertyValues[i];
+  Type type = propertyTypes[i]; ...
+} }
```

(a) An added method from the patch of NH-3931. (b) An added method from the patch of HHH-9864.

Fig. 12. A failure due to hunks with only additions.

The threat to internal validity includes possible errors in our manual classification of bugs. As outsiders, we have limited knowledge of the subjects and introduce wrong classification results. We carefully design protocols and report our patches to programmers to reduce this threat. To enhance transparency and allow for independent verification, we also release our inspection results on our website, so other researchers can check our inspection results.

The threat to external validity arises from the limited number of projects. Our focus on Java and C# may restrict the generalizability of our findings to projects in other programming languages. While these two languages represent a significant portion of modern software development, future work should explore additional programming languages.

8 RELATED WORK

Our study is related to the following research themes:

Empirical studies on bugs. Researchers conduct various empirical studies about bug reports. Anvik *et al.* [48] investigate the patterns and the standard life cycle of bug reports. Bettenburg *et al.* [50] chose duplicate reports as their research target. Guo *et al.* [62] propose the factors (*e.g.*, the reputations of the reporter) affecting which bug shall be fixed. Guo *et al.* point out that poor bug reports may lead to a great number of reassignments [63]. Some researchers focus on how to improve the quality of bug reports [49, 111]. Other researchers target more specific topics regarding bugs. Lin *et al* [73] compare two bug assignment approaches. Xia *et al.* [96] investigate bug report field reassignment. Ding *et al* [57] analyze bugs detected by fuzzing from open-source projects. Some empirical studies are about bug fixing time [104, 105], and some investigate bug fixing in open source projects [51, 59]. Some researchers focus on bugs in specific systems, *e.g.* deep learning frameworks [88, 89, 102, 106], industrial financial systems [98], and smart contracts projects [92]. Mondal *et al.* conduct studies on bug propagation through code cloning [65, 76]. Li and Zhong [71] conduct an empirical study on the impacts of obsolete bug fixes. Non-reproducible bugs are also analyzed in some recent studies [60, 86]. Yan *et al.* [100] report that many bug fixes are workarounds. Our work is the first study on mirror bugs, complementing the above studies.

Detecting cross-language clones. A few approaches are proposed to detect cross-language clones. Based on the .NET intermediate representation (IR), Kraft *et al.* propose the first approach called *C2D2* [69] to detect clones between C# and Visual Basic code, and Al-Omari *et al.* [44] propose an approach to detect clones in more .NET languages. Many programming languages do not have such IRs. For such languages, Cheng *et al.* propose the first approach called *CLCMiner* [56] to detect clones by analyzing revision histories. Vislavski *et al.* propose *LICCA* [91] to use high-level representations to detect cross-language clones. Nichols *et al.* [82] propose an approach that combines the similarity of both code structures and identifiers. Nafi *et al.* [77] propose *CLCDSA* that trains DNN models based on extracted syntactic features. Perez *et al.* [84] and Yahya *et al.* [99] use abstract syntax trees (ASTs) as input to train their models. Tao *et al.* [90] propose *C4* that tunes the pre-trained model *CodeBERT* to detect cross-language clones. The above approaches make it feasible to implement a tool for detecting mirror bugs based on our results.

Cross-language code migration. To support the migration across languages, researchers proposed various approaches that mine API mappings. Zhong *et al.* propose MAM [108], which utilizes heuristics on textual or structural similarity to mine API mappings from translated bilingual projects. Nguyen *et al.* present StaMiner [78] extend their MAM and use a statistical model to learn the mappings. Gu *et al.* [61] propose DeepAM that uses sequence-to-sequence learning to support cross-language API migrations. Nguyen *et al.* [79] propose API2API that utilizes Word2Vec to mine API mappings. Bui *et al.* [52] use generative adversarial networks (GAN) to reduce the amount of training data. Some unsupervised learning methods [54, 58] are proposed to mine analogical APIs. As the models are language agnostic, they have the potential to be applied to cross-language tasks. Besides APIs, Karaivanov *et al.* [66] and Nguyen *et al.* [81] propose approaches that treat source code as a sequence of tokens and apply statistical machine translation (SMT) models to translate code. Qin *et al.* [85] propose an approach to infer the UI event mappings between Android and iOS. Our study highlights the importance of the above approaches, since we find that many source files are not migrated due to the huge effort of manual migration.

Cross-language fault localization. Researchers have proposed various approaches to locate the faulty files of a bug report [43]. These approaches compare source files with a bug report and determine similar source files as the faulty files of the bug report. Source files can be implemented in multiple languages, and bug reports can be written in different natural languages. Some researchers consider the above issue, and work on a subfield called cross-language fault localizations. For instance, Xia *et al.* [95] translate bug reports to English and then compare translated bug reports to locate their faulty files. Yang *et al.* [101] customize position encoding and facilitate attention mechanisms to locate cross-language faulty files. Chakraborty *et al.* [53] employ dynamic chunking to segment source code and utilize hard example learning for model fine-tuning. In our study, we find some two-side bugs with bug reports. As the same bugs are implemented in different languages, these bug reports can be used as labeled data to train their models.

9 CONCLUSION AND FUTURE WORK

Although researchers have conducted many studies to analyze the characteristics of bugs, to the best of our knowledge, no previous study explores mirror bugs that appear in the different versions of an application. Mirror bugs are important, since many applications have versions in different programming languages. To complement the knowledge about bugs, we conducted the first empirical study on mirror bugs. In particular, we analyzed 638 bugs that were collected from the Java and C# versions of four projects. Based on our study, we conclude that (1) 15.0% of bugs appear in both Java and C# implementations; (2) 76.2% of fixed two-sided bugs are actively identified by programmers; (3) 81.8% of unfixed two-sided bugs can be reproduced in the other implementations; and (4) bugs in a version are useful to detect bugs of its versions in other languages.

In future work, we plan to extend our work from the following perspectives: (1) analyzing mirror bugs in projects that are implemented in more programming languages; (2) exploring the detection techniques of mirror bugs; and (3) exploring the evolution patterns of mirror bugs.

REFERENCES

[1] 2017. https://issues.apache.org/jira/browse/LUCENE-7894.
[2] 2018. https://issues.apache.org/jira/browse/LUCENE-8175.
[3] 2019. https://issues.apache.org/jira/browse/LUCENE-8755.
[4] 2020. https://hibernate.atlassian.net/browse/HHH-14216.
[5] 2020. https://github.com/nhibernate/nhibernate-core/issues/2552.
[6] 2020. https://github.com/hibernate/hibernate-orm/pull/3590.

[7] 2020. https://github.com/nhibernate/nhibernate-core/pull/2576.
[8] 2020. https://logging.apache.org/log4net/.
[9] 2020. https://github.com/NetTopologySuite/NetTopologySuite/issues/380.
[10] 2021. https://github.com/NetTopologySuite/NetTopologySuite/commit/60eed6b2f2b5cfcf01fcc07fdb0cdac40ee44702.
[11] 2021. https://issues.apache.org/jira/browse/LUCENE-9661.
[12] 2021. https://github.com/NetTopologySuite/NetTopologySuite/issues/567.
[13] 2021. https://issues.apache.org/jira/browse/LUCENE-10118.
[14] 2021. https://issues.apache.org/jira/browse/LUCENE-10118.
[15] 2021. https://issues.apache.org/jira/browse/LUCENE-10042.
[16] 2021. https://github.com/apache/lucenenet/issues/492.
[17] 2022. https://locationtech.github.io/jts/.
[18] 2022. https://github.com/NetTopologySuite/NetTopologySuite.
[19] 2022. http://lucenenet.apache.org/.
[20] 2022. https://nhibernate.info/.
[21] 2022. https://hibernate.org/orm/.
[22] 2022. https://lucene.apache.org/.
[23] 2022. https://logging.apache.org/log4j/2.x/.
[24] 2022. https://issues.apache.org/jira.
[25] 2022. https://github.com/locationtech/jts/issues/827.
[26] 2022. https://issues.apache.org/jira/browse/LUCENE-10401.
[27] 2022. https://gradle.org/.
[28] 2022. https://issues.apache.org/jira/browse/LUCENE-10598.
[29] 2022. https://github.com/locationtech/jts/issues/919.
[30] 2022. https://github.com/apache/lucene-solr/pull/824/commits/ccab563122ca33860e0af759acf90b711ba502be.
[31] 2022. https://github.com/apache/lucene-solr/pull/824/commits/3798f3625320877d3085555237983308e113bc57.
[32] 2022. https://github.com/apache/lucenenet/issues/644.
[33] 2022. https://github.com/apache/lucenenet/issues/738.
[34] 2022. Java Language Specifications. https://docs.oracle.com/javase/specs/jls/se18/html/jls-12.html#jls-12.4.
[35] 2024. https://42matters.com.
[36] 2024. https://www.linkedin.com.
[37] 2024. https://42matters.com/how-many-american-mobile-apps-are-available-on-both-ios-and-android.
[38] 2024. https://github.com/apache/lucenenet/issues/964.
[39] 2024. https://github.com/dongle-the-gadget/TestLucene.
[40] 2024. https://nhibernate.jira.com/browse/NH-2011.
[41] 2024. https://github.com/hibernate/hibernate-orm/pull/5230/files.
[42] 2024. https://nhibernate.jira.com/browse/NH-3931.
[43] Pragya Agarwal and Arun Prakash Agrawal. 2014. Fault-localization techniques for software systems: a literature review. *ACM SIGSOFT Software Engineering Notes* 39, 5 (2014), 1–8.
[44] Farouq Al-Omari, Iman Keivanloo, Chanchal K. Roy, and Juergen Rilling. 2012. Detecting Clones Across Microsoft .NET Programming Languages. In *Proc. WCRE*. 405–414.
[45] Wajdi Aljedaani, Meiyappan Nagappan, Bram Adams, and Michael Godfrey. 2019. A comparison of bugs across the ios and android platforms of two open source cross platform browser apps. In *Proc. MOBILESoft*. 76–86.
[46] Aylton Almeida, Laerte Xavier, and Marco Tulio Valente. 2024. Automatic Library Migration Using Large Language Models: First Results. In *Proc. ESEM*. 427–433.
[47] Giuliano Antoniol, Umberto Villano, Ettore Merlo, and Massimiliano Di Penta. 2002. Analyzing cloning evolution in the linux kernel. *Information and Software Technology* 44, 13 (2002), 755–765.
[48] J. Anvik, L. Hiew, and G.C. Murphy. 2006. Who should fix this bug?. In *Proc. ICSE*. 361–370.
[49] Nicolas Bettenburg, Sascha Just, Adrian Schröter, Cathrin Weiss, Rahul Premraj, and Thomas Zimmermann. 2008. What makes a good bug report?. In *Proc. ESEC/FSE*. 308–318.
[50] Nicolas Bettenburg, Rahul Premraj, Thomas Zimmermann, and Sunghun Kim. 2008. Duplicate bug reports considered harmful? really?. In *Proc. ICSM*. 337–345.
[51] Pamela Bhattacharya, Liudmila Ulanova, Iulian Neamtiu, and Sai Charan Koduru. 2013. An empirical analysis of bug reports and bug fixing in open source android apps. In *Proc. CSMR*. 133–143.
[52] Nghi DQ Bui, Yijun Yu, and Lingxiao Jiang. 2019. SAR: learning cross-language API mappings with little knowledge. In *Proc. ESEC/FSE*. 796–806.
[53] Partha Chakraborty, Mahmoud Alfadel, and Meiyappan Nagappan. 2024. BLAZE: Cross-Language and Cross-Project Bug Localization via Dynamic Chunking and Hard Example Learning. *arXiv preprint arXiv:2407.17631* (2024).

[54] Chunyang Chen, Zhenchang Xing, Yang Liu, and Kent Ong Long Xiong. 2019. Mining likely analogical apis across third-party libraries via large-scale unsupervised api semantics embedding. *IEEE Transactions on Software Engineering* 47, 3 (2019), 432–447.

[55] Honghao Chen, Ye Tang, and Hao Zhong. 2024. An Empirical Study on Cross-language Clone Bugs. In *Proc. ICSE.* 280–281.

[56] Xiao Cheng, Zhiming Peng, Lingxiao Jiang, Hao Zhong, Haibo Yu, and Jianjun Zhao. 2016. Mining revision histories to detect cross-language clones without intermediates. In *Proc. ASE.* 696–701.

[57] Zhen Yu Ding and Claire Le Goues. 2021. An empirical study of oss-fuzz bugs. In *Proc. MSR.* 131–142.

[58] Jan Eberhardt, Samuel Steffen, Veselin Raychev, and Martin Vechev. 2019. Unsupervised learning of API aliasing specifications. In *Proc. PLDI.* 745–759.

[59] Chiara Francalanci and Francesco Merlo. 2008. Empirical analysis of the bug fixing process in open source projects. In *Proc. OSS.* 187–196.

[60] Anjali Goyal and Neetu Sardana. 2019. An empirical study of non-reproducible bugs. *International Journal of System Assurance Engineering and Management* 10 (2019), 1186–1220.

[61] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. 2017. DeepAM: Migrate APIs with Multi-modal Sequence to Sequence Learning. In *Proc. IJCAI.* 3675–3681.

[62] Philip J Guo, Thomas Zimmermann, Nachiappan Nagappan, and Brendan Murphy. 2010. Characterizing and predicting which bugs get fixed: an empirical study of Microsoft Windows. In *Proc. ICSE.* 495–504.

[63] Philip J Guo, Thomas Zimmermann, Nachiappan Nagappan, and Brendan Murphy. 2011. Not my bug! and other reasons for software bug report reassignments. In *Proc. CSCW.* 395–404.

[64] Yoshiki Higo, Shinpei Hayashi, Hideaki Hata, and Meiyappan Nagappan. 2020. Ammonia: an approach for deriving project-specific bug patterns. *Empirical Software Engineering* 25, 3 (2020), 1951–1979.

[65] Judith F. Islam, Manishankar Mondal, and Chanchal K. Roy. 2016. Bug Replication in Code Clones: An Empirical Study. In *Proc. SANER.* 68–78.

[66] Svetoslav Karaivanov, Veselin Raychev, and Martin Vechev. 2014. Phrase-based statistical translation of programming languages. In *Proc. SPLASH.* 173–184.

[67] Miryung Kim, Vibha Sazawal, David Notkin, and Gail Murphy. 2005. An empirical study of code clone genealogies. In *Proc. ESEC/FSE.* 187–196.

[68] Sunghun Kim, Kai Pan, and EE James Whitehead Jr. 2006. Memories of bug fixes. In *Proc. FSE.* 35–45.

[69] Nicholas A. Kraft, Brandon W. Bonds, and Randy K. Smith. 2008. Cross-language Clone Detection. In *Proc. SEKE.* 54–59.

[70] K. Krippendorff. 2011. Computing Krippendorff's Alpha-Reliability. (2011).

[71] Zexuan Li and Hao Zhong. 2021. An empirical study on obsolete issue reports. In *Proc. ASE.* 1317–1321.

[72] Guangtai Liang, Jian Wang, Shaochun Li, and Rong Chang. 2014. Patbugs: A pattern-based bug detector for cross-platform mobile applications. In *Proc. MS.* 84–91.

[73] Zhongpeng Lin, Fengdi Shu, Ye Yang, Chenyong Hu, and Qing Wang. 2009. An empirical study on bug assignment automation using Chinese bug data. In *Proc. ESEM.* 451–455.

[74] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F Bissyandé. 2019. TBar: Revisiting template-based automated program repair. In *Proc. ISSTA.* 31–42.

[75] Matias Martinez and Sylvain Lecomte. 2017. Towards the quality improvement of cross-platform mobile applications. In *Proc. MOBILESoft.* 184–188.

[76] Manishankar Mondal, Banani Roy, Chanchal K. Roy, and Kevin A. Schneider. 2019. An empirical study on bug propagation through code cloning. *J. Syst. Softw.* 158 (2019).

[77] Kawser Wazed Nafi, Tonny Shekha Kar, Banani Roy, Chanchal K. Roy, and Kevin A. Schneider. 2019. CLCDSA: Cross Language Code Clone Detection using Syntactical Features and API Documentation. In *Proc. ASE.* 1026–1037.

[78] Anh Tuan Nguyen, Hoan Anh Nguyen, Tung Thanh Nguyen, and Tien N Nguyen. 2014. Statistical learning approach for mining API usage mappings for code migration. In *Proc. ASE.* 457–468.

[79] Trong Duc Nguyen, Anh Tuan Nguyen, Hung Dang Phan, and Tien N Nguyen. 2017. Exploring API embedding for API usages and applications. In *Proc. ICSE.* 438–449.

[80] 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 *Proc. ICSE.* 315–324.

[81] Anh Tuan Nguynguyenen, Tung Thanh Nguyen, and Tien N Nguyen. 2013. Lexical statistical machine translation for language migration. In *Proc. ESEC/FSE.* 651–654.

[82] Lawton Nichols, Mehmet Emre, and Ben Hardekopf. 2019. Structural and nominal cross-language clone detection. In *Proc. FASE.* 247–263.

[83] Jeremy R Pate, Robert Tairas, and Nicholas A Kraft. 2013. Clone evolution: a systematic review. *Journal of software: Evolution and Process* 25, 3 (2013), 261–283.

[84] Daniel Perez and Shigeru Chiba. 2019. Cross-language clone detection by learning over abstract syntax trees. In *Proc. MSR*. 518–528.

[85] Xue Qin, Hao Zhong, and Xiaoyin Wang. 2019. Testmig: Migrating gui test cases from ios to android. In *Proc. ISSTA*. 284–295.

[86] Mohammad Masudur Rahman, Foutse Khomh, and Marco Castelluccio. 2022. Works for Me! Cannot Reproduce - A Large Scale Empirical Study of Non-reproducible Bugs. *Empir. Softw. Eng.* 27, 5 (2022), 111.

[87] Dhavleesh Rattan, Rajesh Bhatia, and Maninder Singh. 2013. Software clone detection: A systematic review. *Information and Software Technology* 55, 7 (2013), 1165–1199.

[88] Xiaobing Sun, Tianchi Zhou, Gengjie Li, Jiajun Hu, Hui Yang, and Bin Li. 2017. An empirical study on real bugs for machine learning programs. In *Proc. APSEC*. 348–357.

[89] Florian Tambon, Amin Nikanjam, Le An, Foutse Khomh, and Giuliano Antoniol. 2024. Silent bugs in deep learning frameworks: an empirical study of keras and tensorflow. *Empirical Software Engineering* 29, 1 (2024), 10.

[90] Chenning Tao, Qi Zhan, Xing Hu, and Xin Xia. 2022. C4: Contrastive cross-language code clone detection. In *Proc. ICPC*. 413–424.

[91] Tijana Vislavski, Gordana Rakić, Nicolás Cardozo, and Zoran Budimac. 2018. LICCA: A tool for cross-language clone detection. In *Proc. SANER*. 512–516.

[92] Yilin Wang, Xiangping Chen, Yuan Huang, Hao-Nan Zhu, and Jing Bian. 2022. An Empirical Study on Real Bug Fixes in Smart Contracts Projects. *CoRR* abs/2210.11990 (2022).

[93] W Eric Wong, Vidroha Debroy, Adithya Surampudi, HyeonJeong Kim, and Michael F Siok. 2010. Recent catastrophic accidents: Investigating how software was responsible. In *Proc. SSIRI*. 14–22.

[94] W Eric Wong, Xuelin Li, and Philip A Laplante. 2017. Be more familiar with our enemies and pave the way forward: A review of the roles bugs played in software failures. *Journal of Systems and Software* 133 (2017), 68–94.

[95] Xin Xia, David Lo, Xingen Wang, Chenyi Zhang, and Xinyu Wang. 2014. Cross-language bug localization. In *Proc. ICPC*. 275–278.

[96] Xin Xia, David Lo, Ming Wen, Emad Shihab, and Bo Zhou. 2014. An empirical study of bug report field reassignment. In *Proc. CSMR-WCRE*. 174–183.

[97] Shuai Xie, Foutse Khomh, Ying Zou, and Iman Keivanloo. 2014. An empirical study on the fault-proneness of clone migration in clone genealogies. In *Proc. CSMR-WCRE*. 94–103.

[98] Xiao Xuan, Xiaoqiong Zhao, Ye Wang, and Shanping Li. 2015. An empirical study of bugs in industrial financial systems. *IEICE TRANSACTIONS on Information and Systems* 98, 12 (2015), 2322–2327.

[99] Mohammad A. Yahya and Dae-Kyoo Kim. 2022. Cross-Language Source Code Clone Detection Using Deep Learning with InferCode. *CoRR* abs/2205.04913 (2022). https://doi.org/10.48550/arXiv.2205.04913

[100] Aoyang Yan, Hao Zhong, Daohan Song, and Li Jia. 2023. How do programmers fix bugs as workarounds? An empirical study on Apache projects. *Empirical Software Engineering* 28, 4 (2023), 96.

[101] Haoran Yang, Yu Nong, Tao Zhang, Xiapu Luo, and Haipeng Cai. 2024. Learning to Detect and Localize Multilingual Bugs. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 2190–2213.

[102] Yilin Yang, Tianxing He, Zhilong Xia, and Yang Feng. 2022. A comprehensive empirical study on bug characteristics of deep learning frameworks. *Information and Software Technology* 151 (2022), 107004.

[103] Ruru Yue, Na Meng, and Qianxiang Wang. 2017. A characterization study of repeated bug fixes. In *Proc. ICSME*. 422–432.

[104] Feng Zhang, Foutse Khomh, Ying Zou, and Ahmed E Hassan. 2012. An empirical study on factors impacting bug fixing time. In *Proc. WCRE*. 225–234.

[105] Hongyu Zhang, Liang Gong, and Steve Versteeg. 2013. Predicting bug-fixing time: an empirical study of commercial software projects. In *Proc. ICSE*. 1042–1051.

[106] Yuhao Zhang, Yifan Chen, Shing-Chi Cheung, Yingfei Xiong, and Lu Zhang. 2018. An empirical study on tensorflow program bugs. In *Proc. ISSTA*. 129–140.

[107] Hao Zhong, Suresh Thummalapenta, and Tao Xie. 2013. Exposing behavioral differences in cross-language API mapping relations. In *Proc. ETAPS/FASE*. 130–145.

[108] Hao Zhong, Suresh Thummalapenta, Tao Xie, Lu Zhang, and Qing Wang. 2010. Mining API mapping for language migration. In *Proc. ICSE*. 195–204.

[109] Hao Zhong, Xiaoyin Wang, and Hong Mei. 2022. Inferring bug signatures to detect real bugs. *IEEE Transactions on Software Engineering* 48, 2 (2022), 571–584.

[110] Bo Zhou, Iulian Neamtiu, and Rajiv Gupta. 2015. A cross-platform analysis of bugs and bug-fixing in open source projects: Desktop vs. android vs. ios. In *Proc. EASE*. 1–10.

[111] Thomas Zimmermann, Rahul Premraj, Nicolas Bettenburg, Sascha Just, Adrian Schroter, and Cathrin Weiss. 2010. What makes a good bug report? *IEEE Transactions on Software Engineering* 36, 5 (2010), 618–643.