An Empirical Study on Obsolete Issue Reports

Zexuan Li
Department of Computer Science and Engineering
Shanghai Jiao Tong University, China
lizx_17@sjtu.edu.cn

Hao Zhong
Department of Computer Science and Engineering
Shanghai Jiao Tong University, China
zhonghao@sjtu.edu.cn

***Abstract*—Issue reports record valuable maintenance details. Developers write issue numbers into code comprehension and researchers mine knowledge from issue reports to assist various programming tasks. Although issue reports are useful, some of them can be obsolete, in that their corresponding commits are overwritten or rolled back, with the evolution of software. The obsolete issue reports can invalidate their references and descriptions, and can have far-reaching impacts on the approaches built on them. To explore their impacts, we conduct the first empirical study to analyze obsolete issue reports.**

To measure how an issue report becomes obsolete, we define a obsolete ratio of an issue report as its deleted lines over all its modified lines. To support our analysis, we build a tool, ICLinker, that builds the links between an issue report and its commits, and calculates the obsolete ratio for each issue report. In our study, we analyze 70,180 commits and 46,257 issue reports that are collected from 5 Apache projects. We explore two research questions, which concern the distributions of obsolete issue reports and the obsolete references in code comments. Our findings to these research questions enrich the knowledge on obsolete issue reports, and some are even counterintuitive. For example, we find that obsolete issue reports are mixed with other issue reports. Even when recent issue reports are obsolete, some old issue reports keep up-to-date.

I. Introduction

During software maintenance, programmers resolve many issues (*e.g.*, bugs and feature requests). To manage these issues, they describe their symptoms, reproduce steps, expected behaviors and report them on issue tracking systems [3]. As issue reports record many development details, researchers have used issue reports as their data sources in various research tools (*e.g.*, bug report summarization [10], bug assignments [2], fault localization [15] and patch generation [9]).

As issue reports are quite useful to understand the functionalities and purposes of source code, in many projects, programmers are required to manually link issue reports to source files. To help understanding code, some programmers even write issue numbers in their code comments. With the evolution of software, the added lines from an issue report can be overwritten by later commits, and this issue report becomes obsolete. For example, Figure 1a shows the modified lines of `HIVE-7421`, and Figure 1b shows a code comment that refers to this issue report and explains why constant folding is not allowed. We check the latest files, and as shown in Figure 1c, most code lines of the modified file (`VectorUDFDateString.java`) are deleted. As a result, the added code lines of `HIVE-7421` no longer appear in this source file. As we are, other programmers can be confused, when they find the mentioned issue report is obsolete.

```
--- .../VectorUDFDateString.java
+++ .../VectorUDFDateString.java ...
@@ -41,13 +45,10 @@
-   Date date = Date.valueOf(s.toString());
-   t.set(date.toString());
+   Date date = formatter.parse(s.toString());
+   t.set(formatter.format(date));
    return t;...
```

(a) The modified lines of `HIVE-7421`

```
public class VectorizationContext { ...
 private VectorExpression getCastToChar(...)...{ ...
  if (child instanceof ExprNodeConstantDesc) {
// Don't do constant folding here.  Wait until the
     optimizer is changed to do it.
// Family of related JIRAs: HIVE-7421, HIVE-7422, and
     HIVE-7424.
     return null;
  }
 }}
```

(b) A code comment that mentions `HIVE-7421`

```
public class VectorUDFDateString extends
     CastStringToDate {
  private static final long serialVersionUID = 1L;
  public VectorUDFDateString() {}
  public VectorUDFDateString(int inputColumn, int
       outputColumnNum) {
    super(inputColumn, outputColumnNum);
  }
}
```

(c) The latest code of `VectorUDFDateString`

Fig. 1: A obsolete issue report

Obsolete issue reports can have far-reaching impacts on downstream research. For example, to locate the faulty files of a new received bug report, Zhou *et al.* [15] compare the new bug report to the previous ones and recommend the revised files of a similar previous report as the buggy file. If a bug report is similar to `HIVE-7421`, it can recommend `VectorUDFDateString.java` as the faulty file, but as this file now has only several lines of code, it is unlikely to be buggy.

The example shown in Figure 1 is common, and a recent study [4] shows that the added lines of some programmers are all rewritten by other programmers. Although obsolete issue reports can influence both programmers and researchers, no prior researches have made empirical study to our best knowledge, and many questions on obsolete issue reports are still open. For example, to what extent issue reports

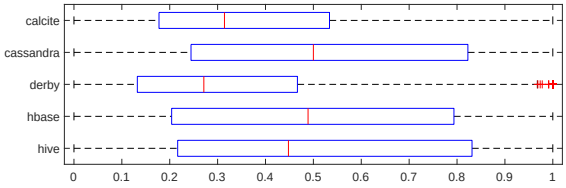


Fig. 2: The distributions of obsolete ratios

are obsolete, and which types of issue reports contain more obsolete ones?

To answer these questions, we build a tool, ICLINKER, that calculates the obsolete ratios for issue reports. ICLINKER links commits of a project to issue reports from the issue tracker system. It compares the commits with its corresponding latest source file, and calculates an obsolete ratio for each issue report. The obsolete ratio is calculated as the proportion of its added lines that cannot match to the latest source files. With ICLINKER, we answer the following research questions:

RQ1: What is the distribution of obsolete issue reports? By calculating the obsolete ratio for each issue report, we draw box plots to show the distribution. With the distributions, researcher can prevent the impacts from obsolete issue reports to their researches.

RQ2: How do code comments mention obsolete issues? We explore obsolete issue references that appear in code comments. With the findings, developers can tune the maintenance process and keep the consistency of code and comment.

II. METHODOLOGY

This section introduces our methodology. In this study, we define the following terms:

Definition 1 (Obsolete issue report): For an issue report (r), if the added lines in its revisions are partially or totally removed in later commits, we treat the issue report as obsolete.

To determine to what degree the issue report is obsolete, for each modified file of this report, we calculate an obsolete ratio as follows:

Definition 2 (Obsolete ratio): For an issue report (r), the obsolete ratio is the proportion of its added lines that cannot match to the latest source files. It is calculated as:

$$ratio = 1 - \frac{r_{al}}{r_a} \quad (1)$$

where r_{al} denotes the added lines that appear in the latest source files and r_a denotes all the added line of r.

For an issue report with multiple file modified, we calculate its obsolete ratio as the average ratios of all its modified files. If a past bug report is unlinked, this report has no faulty files, and it is infeasible to use the report, and it becomes less interesting. In such situations, unlinked issue reports are close to totally obsolete ones, so we set their ratios as 1.

A. ICLINKER

ICLINKER extracts commits from code repositories (Section II-A1); links commits and issue reports (Section II-A2); and compares the latest source files with commits. It thus calculates an obsolete ratio for each issue report (Section II-A3).

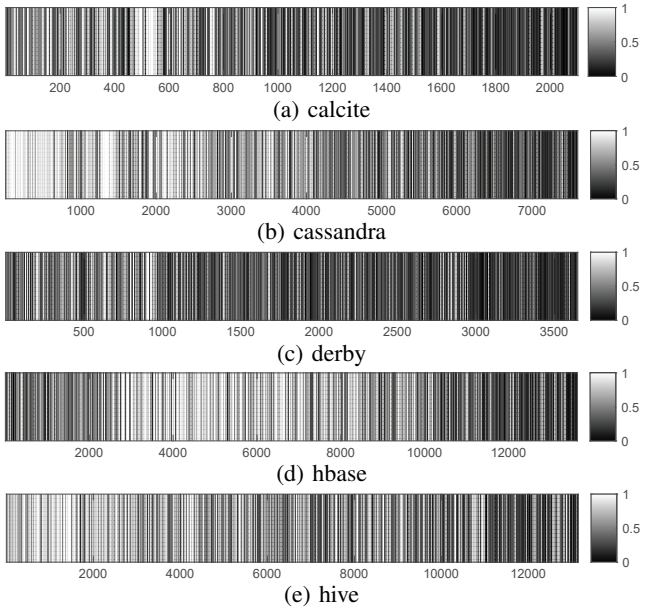


Fig. 3: The obsolete issue reports in the chronological order

1) Extracting Commits: ICLINKER is built on a version-control library called `JGit` [1]. `JGit` provides APIs to retrieve and parse commits of a repository. ICLINKER first extracts commits of all files. After that, for each source file in the latest version, it traverses the tree to identify all the commits on this file. For each commit, ICLINKER extracts added lines (*i.e.*, the lines starting with "+" in Figure 1a).

2) Linking Commits and Issue Reports: Le *et al.* [7] reported that in Apache projects, most revisions explicitly mention the corresponding issue reports in their commit messages. Therefore, ICLINKER uses this pattern to link commits to issue reports as the prior studies [3], [7] did. Thus, given a source file in the latest version, ICLINKER identifies all its commits, and further rebuilds the links between commits and issue reports. If an issue report has more than one commit, ICLINKER determines that an issue report is obsolete by comparing the modifications of all its commits.

3) Matching Code: To determine to what degree the issue report is obsolete, ICLINKER calculates the obsolete ratio by comparing the latest source file with the added lines of a commit. To save the comparison effort, it analyzes only revisions whose corresponding issue reports are already identified.

III. EMPIRICAL RESULT

In this section, we introduce our protocol for each research question, and our empirical results. More details of our empirical study are listed on our website:
https://github.com/lizx2017/rotten-issue

A. Dataset

Table I shows our dataset. We select these projects, because they are popular and under active maintenance. All our selected projects are downloaded from the Apache foundation, because most Apache projects carefully write issue numbers in commit messages.

TABLE I: The dataset and our generated comments.

Project	Dataset						Issue References								
	Commit			Issue			Issue reference in code comment						Total fixed issue		
	linked	total	%	linked	total	%	bug	%	feature	%	other	%	bug	feature	other
calcite	2,363	4,188	56.4%	2,101	2,608	80.6%	403	23.3%	56	9.3%	15	5.4%	1,727	605	276
cassandra	19,724	25,299	78.0%	7,604	8,931	85.1%	238	4.6%	81	2.7%	16	2.2%	5,181	3,032	718
derby	6,993	8,269	84.6%	3,650	4,586	79.6%	900	34.2%	218	16.3%	50	8.1%	2,629	1,339	618
hbase	16,441	17,794	92.4%	13,636	16,200	84.2%	320	4.1%	114	3.0%	142	3.1%	7,869	3,739	4,592
hive	14,208	14,630	97.1%	13,149	13,932	94.4%	150	1.9%	51	1.9%	79	2.2%	7,713	2,681	3,538
total	59,729	70,180	85.1%	40,140	46,257	86.8%	2,011	8.0%	520	4.6%	302	3.1%	25,119	11,396	9,742

linked: the numbers of commits/issues who have corresponding links to issues/commits by our name heuristic.
feature: improvements and new features; bug: bug reports; other: remaining

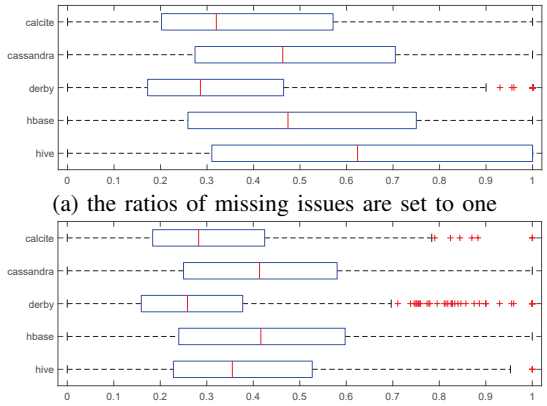
B. RQ1. Overall Distribution over Time

1) Protocol: For each issue report, we use ICLINKER to calculate its obsolete ratios. A box plot is a diagram that displays the median, upper and lower bounds of a data set. First, we classify obsolete ratios by their projects, and we draw box plots to show the distributions over projects. To present vivid distributions, for each project, we then rank its issue reports by their numbers, and draw a graph to show all obsolete ratios, since the issue numbers of issue reports increase over time. Although box plots are informative, they are intuitive, and they do not present the distribution over time directly.

2) Result: Figure 2 shows the overall distributions. For `cassandra`, `hbase` and `hive`, the medians are around 0.5, and for `calcite` and `derby`, the ratios are around 0.3. Table I shows that `calcite` and `derby` have much fewer commits and issue reports than the other three projects. However, the Pearson product-moment correlation coefficient calculation between obsolete ratios and commits is 0.17, and the result indicates obsolete ratios have only a weak correlation with the number of commits. As the other three projects have longer histories of evolution, more issue reports of the three projects become obsolete than those of `calcite` and `derby`. The above observations lead to the first finding:

> **Finding 1:** *A project with more issue reports typically has a larger ratio of obsolete issue reports. If a project has a long history, at least half of its issue reports are becoming obsolete.*

Figure 3 shows all the obsolete ratios for each project. A lighter color denotes a more obsolete issue report. All the issue reports are sorted by their issue numbers, and a larger value of the horizontal axis indicates a larger issue number. The lines of each project construct a bar. In Figure 3, for `calcite`, `cassandra`, `derby`, and `hive`, the left sides are lighter than the right sides, indicating that the older issue reports are more obsolete. The result of `hbase` follows a different pattern, and its middle issue reports are lighter. Besides, we notice that the line colors do not change smoothly. For all the projects, some lines on the left side are black, while some lines on the right are lighter. It means that the obsolete ratios of issue reports are not linear with the issue numbers. Instead, obsolete issues are mixed with up-to-date ones. The above observations lead to a finding:



(a) the ratios of missing issues are set to one

(b) missing issues are ignored

Fig. 4: Obsolete issue references in comments

> **Finding 2:** *As a trend, a more recent issue report often has a lower obsolete ratio. However, obsolete issue reports are mixed with other issue reports.*

C. RQ2. Obsolete Issue References in Code Comments

1) Protocol: In this research question, we analyze the issue references that appear in code comments. We extract issue numbers from code lines and Table I shows the results. Among the fixed issue reports, the number of bug reports is the largest. Accordingly, among the issue references, the number of bug references is the largest, except `hive`. Compared with the other three projects, `calcite` and `derby` mention much more issue reports in their code comments.

As we do in Section III-B1, we draw box plots and image graphs to show the distributions of obsolete ratios of the issue references that appear in code comments. Since a comment can mention an unresolved issue or an issue whose commits are not recovered, we provide two strategies to handle such cases. In the first strategy, we set their obsolete ratios as one. As it is infeasible to locate their modifications, these issue references are similar to totally obsolete ones. In the second strategy, we ignore all such issue references. We present the results for both strategies.

2) Result: Figure 4 shows the distributions of obsolete ratios. Compared with the results in Figure 2, Figure 4 shows that issue references in code comments have similar distributions. In Figure 4, the medians of `calcite`, `cassandra`, `derby`, and `hbase` are close to those of Figure 2. The code

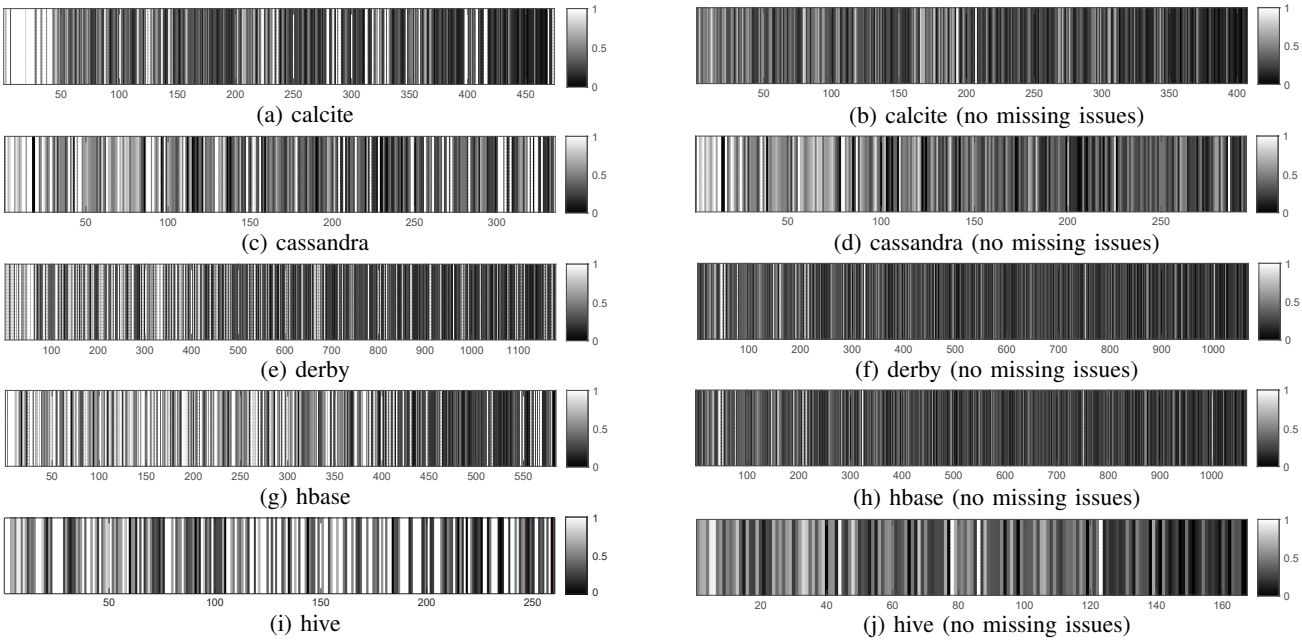


Fig. 5: Obsolete issue references in the chronological order

comments of `cassandra` and `hbase` mention more stable issue reports, and their upper bounds are reduced from more than 0.8 to around 0.7. However, the median and the upper bound of `hive` both become larger than those of Figure 2.

Figure 5 shows the obsolete issue references of code comments in the chronological order. Compared with the results in Figure 3, the obsolete references in code comments are also twisted with those fresh issue references. The above observations lead to a finding:

> **Finding 3:** *Code comments contain many references to obsolete issue reports, and such references are twisted with references to fresh issue reports.*

Figure 5i is quite different from Figure 5j. We find that its programmers often write issue numbers in the to-do lists of code comments:

```
1 private ValidTxnWriteIdList getQueryValidTxnWriteIdList
      () ... {
2   // TODO: Once HIVE-18948 is in, should be able to
        retrieve writeIdList from the conf.
3   //cachedWriteIdList = AcidUtils.getValidTxnWriteIdList
        (conf);
4   ...}
```

In the above code, Line 2 mentions `HIVE-18948`. This issue report was filed in 2018, but it is still unresolved. The `hive` projects have more such cases than others.

D. Threat to Validity

The threat to internal validity includes our definition of the obsolete ratio. Even if a ratio is low, the modifications on several important statements make an issue report more obsolete than it appears to be. The threats to external validity include our subjects. The generality of our results could be improved, if more projects and projects from more sources are analyzed. As projects from other sources may not be carefully maintained, we need more advanced techniques [12] to build the links between issue reports and commits.

IV. Work Plan

To extend to full paper, our work plan is:

1. Exploring which factors contribute to the obsolesce of issue reports. We plan to explore which factors contribute to the obsolesce of issue categories. For example, the issue reports of a specific component can have more obsolete issue reports than others. The factors can be useful to understand the obsolesce of issue reports.

2. Cleansing the inputs of prior approaches. Researchers have proposed approaches to mine knowledge from issue reports. Their mined knowledge is used to configure tools [13], to predict bug severity [14], and to tune development processes [5]. Obsolete issue reports can invalid some knowledge, but the prior approaches do not take them into consideration. ICLinker can be used to cleanse their inputs, and improve their effectiveness. When we use issue reports to generate code comments [8], we already considered obsolete issue reports, and have removed them from our inputs.

3. Detecting wrong issue references. It is difficult to keep the references between software artifacts consistent. For example, even if code comments are written near implementations, they can become inconsistent [6], [11]. When issue reports are obsolete, their references can become incorrect. A detection tool can be useful to maintain issue reports more effectively.

Acknowledgment

We appreciate the anonymous reviewers for their insightful comments. Hao Zhong is the corresponding author. This work is sponsored by the National Key R&D Program of China No.2018YFC083050.

REFERENCES

[1] JGit. https://www.eclipse.org/jgit/.

[2] J. Anvik, L. Hiew, and G. Murphy. Who should fix this bug? In *Proc. 28th ICSE*, pages 361–370, 2006.

[3] T. F. Bissyandé, D. Lo, L. Jiang, L. Réveillère, J. Klein, and Y. L. Traon. Got issues? who cares about it? a large scale investigation of issue trackers from github. In *Proc. ISSRE*, pages 188–197, 2013.

[4] S. Gong and H. Zhong. Code authors hidden in file revision histories: An empirical study. In *Proc. ICPC*, pages 71–82, 2021.

[5] M. Gupta and A. Sureka. Nirikshan: Mining bug report history for discovering process maps, inefficiencies and inconsistencies. In *Proc. MSR*, pages 1–10, 2014.

[6] Z. M. Jiang and A. E. Hassan. Examining the evolution of code comments in postgresql. In *Proc. MSR*, pages 179–180, 2006.

[7] T. B. Le, M. Linares-Vasquez, D. Lo, and D. Poshyvanyk. Rclinker: Automated linking of issue reports and commits leveraging rich contextual information. In *Proc. ICPC*, pages 36–47, 2015.

[8] Z. Li and H. Zhong. Understanding code fragments with issue reports. In *Proc. ASE*, page to eappear, 2021.

[9] C. Liu, J. Yang, L. Tan, and M. Hafiz. R2fix: Automatically generating bug fixes from bug reports. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, pages 282–291, 2013.

[10] S. Mani, R. Catherine, V. S. Sinha, and A. Dubey. Ausum: approach for unsupervised bug report summarization. In *Proc. FSE*, pages 1–11, 2012.

[11] F. Wen, C. Nagy, G. Bavota, and M. Lanza. A large-scale empirical study on code-comment inconsistencies. In *Proc. ICPC*, pages 53–64, 2019.

[12] R. Wu, H. Zhang, S. Kim, and S.-C. Cheung. Relink: recovering links between bugs and changes. In *Proc. ESEC/FSE*, pages 15–25, 2011.

[13] X. Xia, D. Lo, W. Qiu, X. Wang, and B. Zhou. Automated configuration bug report prediction using text mining. In *Proc. COMPSAC*, pages 107–116, 2014.

[14] T. Zhang, G. Yang, B. Lee, and A. T. Chan. Predicting severity of bug report by mining bug repository with concept profile. In *Proc. SAC*, pages 1553–1558, 2015.

[15] J. Zhou, H. Zhang, and D. Lo. Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports. In *Proc. ICSE*, pages 14–24, 2012.