

Understanding Code Fragments with 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

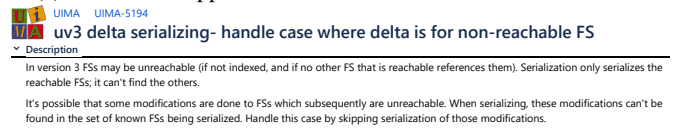
Abstract—Code comments are vital for software development and maintenance. To supplement the code comments, researchers proposed various approach that generate code comments. The prior approaches take three sources: (1) programming experience, (2) code-comment pairs in source files, and (3) comments of similar code snippets. Most of their generated comments explain code functionalities, but programmers also need comments that explain why a code fragment was developed as it is. To meet the timely needs, in this paper, we introduce a new source, issue reports (e.g. maintenance types, symptoms, and purposes of modifications), to generate code comments. Issue reports contain rich information on how code was maintained. The valuable details of issue reports are useful to understand source code, especially when programmers learn why code was developed in a specific way. Towards this research direction, we propose the first approach, called ISSUECOMM, that builds the links between code fragments and issue reports. Our results show that it links more than 70% issue numbers that are written by programmers in code comments. Furthermore, the links built by our tool covers 4× bugs, and 10× other issues than the links written in manual comments. We present samples of our built links, and explain why our links are useful to describe the functionalities and the purpose of code.

I. INTRODUCTION

Code comments are helpful for program comprehension. When developers fix a bug whose code was developed a long time ago, to understand its functionalities, they often read code comments before repairing the bug. However, in practice, code comments are insufficient and noisy due to poor programming habits. To attack this problem, researchers proposed various approaches that generate code comments. Based on their inputs, the prior approaches can be roughly divided into three categories such as template-based approaches, learning-based approaches, and clone-based approaches. Taking the programming experience of researchers as their inputs, template-based approaches define various templates to generate comments for methods [16], classes [14], thrown exceptions [4] and test files [22]. Taking code-comment pairs in source files as their inputs, learning-based approaches reduce generating code comments to a sequence-to-sequence problem [17]. For this problem, researchers resolve it with RNN [9], encoder [5], and the convolutional attention network [3], and they introduce API sequences [8] and ASTs [7] to improve the quality of generated comments. Clone-based approaches reuse comments from similar code snippets from Stackoverflow threads [20] and code clones [19]. The above approaches typically generate comments that explain the functionalities of code fragments. When programmers maintain unfamiliar source files, they need

```
1 private void serializeModifiedFSs() throws IOException {
2     final int addr = csds.fs2addr.get(fs);
3     if (addr == 0) {
4         // https://issues.apache.org/jira/browse/UIMA-5194
5         // need to write a dummy entry because...
6         writeVnumber(fsIndexes_dos, 0); }...}
```

(a) The code snippet of `serializeModifiedFSs()` in UIMA



UIMA UIMA-5194
uv3 delta serializing- handle case where delta is for non-reachable FS
Description
In version 3 FSs may be unreachable (if not indexed, and if no other FS that is reachable references them). Serialization only serializes the reachable FSs; it can't find the others.
It's possible that some modifications are done to FSs which subsequently are unreachable. When serializing, these modifications can't be found in the set of known FSs being serialized. Handle this case by skipping serialization of those modifications.

(b) The issue report (modified to save space)

Fig. 1: A link in a code comment

more insightful comments. However, from source files or code clones, it is rather challenging to learn such comments. For example, to explain why a code fragment is implemented in a specific way, a programmer must be quite familiar with this fragment and the whole system. The knowledge from such a program is difficult to be extract from source files and may not appear in code clones.

To improve the state of the art, in this paper, we identify issue reports as a new source for generating code comments. Issue reports contain many details on maintaining code, and are quite useful to understand code fragments. We observe that programmers have written the issue reports numbers in code comments. For example, without any code comments, it is difficult to understand the purpose of these lines in Figure 1a. The comment in Figure 1a contains a reference to an issue report as shown in Figure 1b. As explained in this issue report, this code fragment handles the situation when `addr` is zero. The descriptions in the issue report make a clear explanation.

Towards this research direction, we propose the first approach, called ISSUECOMM, that links code fragments to their corresponding issue reports. After the links are built, ISSUECOMM implements a simple module to insert links to the code comments of corresponding code fragments. We evaluated ISSUECOMM on seven projects, and the results are promising. On one hand, in all the projects, we find that programmers write some issue numbers in their comments, which highlights the usefulness of linking issue reports to code lines. On the other hand, in all the analyzed projects except derby, programmers write only a small portion of issue numbers in their comments (less than 30%), which indicates that many links are still missing.

TABLE I: Issue numbers in code comments that were written by programmers.

Project	Description	Issue in comment						Total issue			File with issue		Total file
		bug	%	feature	%	other	%	bug	feature	other	code	%	code
activemq	a messaging server	125	4.1%	18	1.2%	3	1.0%	3,067	1,486	303	27	1.5%	1,770
aries	an OSGi library	31	3.5%	7	1.3%	1	1.8%	874	524	56	19	1.7%	1,115
carbondata	a data store	0	0.0%	3	0.4%	0	0.0%	1,467	721	444	2	0.2%	940
cassandra	a partitioned row store	236	4.6%	81	2.7%	14	1.9%	5,181	3,032	718	165	8.5%	1,934
derby	a relational database	911	34.7%	218	16.3%	50	8.1%	2,629	1,339	618	292	16.0%	1,829
mahout	a learning library	8	1.3%	4	0.6%	0	0.0%	620	638	136	10	1.4%	735
uima	a mining library	93	3.2%	31	1.8%	1	0.2%	2,941	1,751	508	87	5.9%	1,469
total	-	1,404	8.4%	362	3.8%	69	2.8%	16,779	9,491	2,434	602	6.1%	9,792

bug: bugs; feature: feature requests and improvements; and other: other types (e.g., tasks). code: source files

In summary, this paper makes the following contributions:

- **A new source for generating comments.** We identified issue reports as a new source for generating code comments. Towards this direction, we implement ISSUECOMM that links issue reports to code lines.
- **Promising early results.** We compare existing comments with the results of ISSUECOMM. the results show that our tool relinks 70% issue numbers that are written in manual comments. In addition, our generated links are 10 times more than those are written in manual comments, and our links are useful to explain code functionalities and implementation rationales.

II. MOTIVATING EXAMPLE

Figure 1a shows a serialization method of `uima`. Line 3 handles the situation when `addr` is zero. If a programmer is unfamiliar with the code, it is difficult to understand why the situation must be handled.

Although various code comment generation approaches can be used, they cannot explain the rationale behind these code snippets. For example, Hu *et al.* [6] train a deep learning model to generate code comments which explain only the functions of code lines. Wong *et al.* [19] recommend comments to a method by searching its code clones, but this method has no clones. As a result, neither approach can build the links of issue reports for this method.

In Figure 1a, the comment of Line 3 presents the url of an issue report. Figure 1b shows the issue report. It depicts the buggy behavior; explains the reasons; and provides the solution. After reading the issue report, we understand that Line 3 fixes serialization bug when a delta is not reachable. Thus, our insight is that linking issue reports to code lines is helpful for code comprehension.

As shown in Figure 1a, programmers can manually link their code lines to issue reports. We conduct an empirical study to explore this practice. Table I shows the results. Column “Issue with comment” lists the numbers and percentages of the comments that contain corresponding types of issue reports. Column “Total issue” lists the total numbers of resolved issue reports. Here, we do not count those duplicated and superficial issue reports (e.g., not a problem). The result shows that the programmers of all the projects write issue numbers in their code comments, and they write issue numbers of bug fixes more frequently than those of other types.

Column “File with issue” lists the numbers and percentages of source files that contain comments with issue numbers. We remove test files. The results show that programmers seldom write down issue numbers into comments of source files.

Although it is beneficial to write issue numbers in code comments, it takes too much manual effort. As a result, in total, only 8.4% issue numbers of bugs are written.

In summary, although it is beneficial to link issue reports to code fragments, due to the huge manual effort, in practice, programmers often write only a small portion of issue numbers in comments. The state of the practice reveals a strong need for an automated tool.

III. APPROACH

Figure 2 shows the overview of ISSUECOMM. It takes a source file and issue reports as its inputs, and builds the links between code fragments and issue reports. It has a revision extractor (Section III-A), an issue matcher (Section III-B), and a code commenter (Section III-C).

A. Extracting Revisions

In a development team, programmers often use version control systems to keep the source file consistent, e.g. *Git*. *Git* stores the revision history of source files in `.git` directory. The version-control library `JGit` provides APIs to parse the repositories and traverses the revisions under the `.git` directories. Built on `JGit` [1], ISSUECOMM first stores revision details into a local database and patches into local directories. After that, for each source file in the latest version, it traverses the tree to identify all the revisions on this file. In this way, for each source file in the latest version, ISSUECOMM extracts all its revisions.

B. Extracting Issues

ISSUECOMM implements a web crawler to extract issue reports from the issue tracking system. For each issue report, ISSUECOMM parses its content and stores its details into a local database. Then ISSUECOMM extracts the links between issue reports and revisions by matching commit messages with issue numbers. Indeed, many Apache projects explicitly define that commit messages shall contain corresponding issue numbers. For example, Apache Geode [2] defines that a commit message shall start with its corresponding issue number. Given a source file in the latest version, our revision extractor in

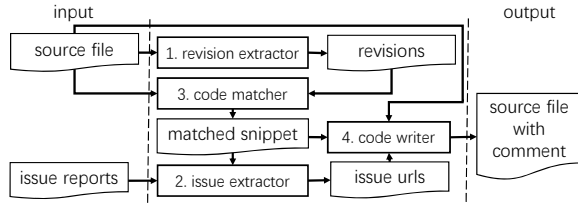


Fig. 2: The overview of ISSUECOMM

Section III-A identifies all its revisions, and in this section, our issue extractor further identifies the links between revisions and issue reports. As a result, given a source file in the latest version, we can identify all of its related issue reports.

C. Commenting Code

Our code commenter compares a latest source file with the patch of a revision to determine whether its issue report shall be linked and where to link the report. In a patch of a revision, lines started with “+” represent the added lines, while the lines started with “-” represent the removed lines. ISSUECOMM identifies and removes useless modified lines, including code comments, import statements, bracket lines, annotation lines, the headers of packages, classes and methods, log lines and variable declarations. Code commenter then generates a comment only when a patch has more than one useful added line. For each remaining patch, it inserts a comment to the first added line, if such a line appears in the latest file.

IV. EVALUATION

With ISSUECOMM, we conduct evaluations to explore the following research questions:

- (RQ1) How many correct links are recovered (Section IV-B)?
 (RQ2) How many more links are recovered (Section IV-C)?

More details of our evaluations are listed on our website:

<https://github.com/lizx2017/issuelinker>

A. Dataset

Table I shows the dataset of our evaluation. We select seven open source projects as our dataset since they are popular, under careful and active maintenance.

B. RQ1. Overall Effectiveness

1) *Setup*: In this research question, we analyze how many links in code comments are identified by ISSUECOMM. As we introduced in Section II, programmers can write issue numbers in code comments. Table I shows the issue numbers in manually written code comments. It is improper to use manual code comments as a gold standard, because they are incomplete. Even if ISSUECOMM generates a correct link, it will be considered as a false positive. However, manual code comments can be used as a reference to our results. Although manual links are incomplete, they shall be correct. We use them as a reference to measure how many correct links are identified by our tool. We calculate our recall as follows:

$$recall = \frac{i_o \wedge i_m}{i_m} \quad (1)$$

where i_m denotes the number of issue reports that are manually written in code comments and $i_o \wedge i_m$ denotes the number of issue reports in both manual code comments and our built links. Similarly, we define the recall for files:

$$recall = \frac{f_o \wedge f_m}{f_m} \quad (2)$$

where f_m denotes the number of files that contain manual comments with issue numbers, and $f_o \wedge f_m$ denotes the number of files with manual links and our built links.

2) *Result*: Table II shows the overall results. Columns “Bug”, “Feature”, and “Other” list the numbers and recalls of the corresponding types of issue reports, and their definitions are the same with Table I. The results show that ISSUECOMM achieves high recalls on *aries*, *derby*, and *umia*. For *cassandra*, the recalls are around 60%. As *carbondata* has only three manual links, its result is not representative, and two of its recalls are not a number, because their manual links have not such types of issue numbers. Only the recalls of *activemq* on “other” are low (around 30%). We check the linked issues of all projects and find that *activemq* has the fewest links between commits and issue reports, which can partially explain its low recalls. In total, for all the types of issue reports, our tool achieves around 75% recalls. As manual comments can mention issue reports for more purpose than their modifications, and manual links are not fully correct, our recalls are already quite high.

In summary, for all types of issue reports, our generated links cover about 70% links that appear in the comments written by programmers.

C. RQ2. The Improvements over Manual Links

1) *Setup*: In this research questions, we compare manual links and our built links to show how we improve the state of the practice. We define the delta of the improvements as:

$$\Delta_i = \frac{i_o}{i_m} \quad (3)$$

Similarly, we define the delta of files as follows:

$$\Delta_f = \frac{f_o}{f_m} \quad (4)$$

The symbols are of the same meanings as Equations 1 and 2.

2) *Result*: Table II shows the results. Columns “ Δ of issue” list the improvements over those of the manual links. The results show that for all the types of issue reports, our built links improve the manual links of *activemq*, *aries*, *carbondata*, and *mahout* by more than ten times. Some deltas of *carbondata* and *mahout* are not a number, because their manual links do not mention any issue numbers, *i.e.*, i_m is zero. Table I shows that *cassandra* and *derby* have more comments with issue numbers, but even for the two projects, we improve their comments by two and more times. Table I shows that programmers write the issue numbers of more bug reports than those of other issue reports. As a result, our improvements on bugs are less than those on other types of issue reports, but in total, the improvement on bugs is still five times more than that of the manual links.

Column “ Δ of file” shows the improvements over those of manual links. The results show that in total, our tool generates

TABLE II: The manual links that were identified by ISSUECOMM.

Project	RQ1						RQ2					
	Bug		Feature		Other		Code		Δ of issue			Δ of file
	no.	%	no.	%	no.	%	no.	%	bug	feature	other	code
activemq	103	82.4%	13	76.5%	1	33.3%	16	59.3%	12.36	43.24	27.33	32.56
aries	24	82.8%	5	83.3%	1	100.0%	11	57.9%	13.97	44.83	25.00	31.63
carbondata	0	n/a	2	66.7%	0	n/a	0	0.0%	n/a	99.00	n/a	354.00
cassandra	146	61.9%	51	63.0%	8	57.1%	74	44.8%	6.07	13.27	12.14	8.73
derby	768	85.3%	175	82.9%	42	84.0%	232	79.5%	1.89	3.95	6.94	6.97
mahout	3	37.5%	1	33.3%	0	n/a	8	80.0%	24.63	86.00	n/a	58.60
uima	76	81.7%	23	74.2%	1	100.0%	70	80.5%	5.02	11.84	57.00	12.79
total	1,120	79.8%	270	74.6%	53	76.8%	411	68.3%	4.39	10.59	12.68	12.23

comments for more than 16 times of source files than those of the manual comments. As an extreme case, in `carbondata`, our tool generates comments for more than one hundred times of files. The results show that our tool significantly improves manual links, because it generates much more (10 × on most projects) links to issue reports.

We next present some sample links to show their usefulness:

1. A link to an improvement.

```

1 protected UnfilteredRowIterator computeNext() {
2     ...
3     try (UnfilteredRowIterator partition = ...) {
4 // see: https://issues.apache.org/jira/browse/CASSANDRA-11183
5         Row staticRow = partition.staticRow();
6         List<Unfiltered> clusters = new ArrayList<>();
7         while (partition.hasNext()) {
8             Unfiltered row = partition.next(); ...
9         } ...

```

The above code has no manual comments, and it is difficult for a novice programmer to understand its functionality. Our tool adds Line 4. This mentions issue report introduces that programmers extend `cassandra` to support static columns. our link is useful for programmers to understand that the follow-up lines handle static columns.

2. A link to a task.

```

1 // see: https://issues.apache.org/jira/browse/AMQ-1846
2 protected Subscription createSubscription(...) ... {
3     ActiveMQDestination destination=info.getDestination();
4     PolicyEntry entry = null; ...

```

A built link on `activemq` is shown as above. The method has no comments, and it is difficult for a novice programmer to understand its functionality. Our tool finds that this method is added in `AMQ-1864`. This issue report is a task, and its title is “Provide tags to set defaultPrefetchSize in `activemq.xml`”. This comment is useful for programmers to understand code.

3. A link to a bug.

```

1 if (isXa) { ...
2 // see: https://issues.apache.org/jira/browse/AMQ-3863
3 } else { session.setIgnoreClose(false); }

```

The above `activemq` bug report complains that a session returns twice from a pool, and the problem exhausts the pool. To fix the bug, Lines 4 to 6 are added to the above code. The added lines set the flag to allow sessions be automatically closed. our link is useful for programmers to understand why it is necessary to call the `setIgnoreClose` method.

In summary, ISSUECOMM generates much more (10 times on most projects) links to issue reports, which significantly improves the existing manual links.

D. Threat to Validity

The external threat to validity includes our subjects. Although we select only Java projects, our approach is general to other languages. The internal threat to validity includes our strategy to match commit messages with issue numbers. The strategy relies on commit messages containing issue numbers. Although software developers record the issue numbers into commit messages in most case, it would be better to introduce alternative tools for commit matching. The internal threat to validity includes our gold standard of our evaluation. We use the manual links in code comments as the gold standard, but such links are neither fully precise nor complete, since it takes too much effort to comment all links and code comments can become insistent with implementations [10], [18]. To reduce the threat, we conduct manual inspections on those inconsistent cases.

V. WORK PLAN

To extend our work to a full paper, our research plan is:

1. Summarizing issue reports. Researchers have proposed approaches to summarize bug reports [11], [15] and code changes [13], [21]. In the future work, we plan to summarize issue reports and measure the quality of our comments.

2. Comparing with prior approaches. As our inputs are different from the prior approaches, it is infeasible to compare them with controlled experiments. However, it is interesting to explore the characteristic of comments that are generated from other sources (e.g., comments that are learnt from code [7] and code clones [19], [20].) As the comparison of classical measures (e.g., f-score) are useful only in controlled experiments, we will explore advanced comparison techniques.

3. Presenting more technical details. Due to space limit, we cannot write all details of our approach. For example, we have removed obsolete issue reports [12], since their links are not built. When extending our work to a full paper, we will provide such details.

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.2018YFC0830500.

REFERENCES

- [1] JGit. <https://www.eclipse.org/jgit/>.
- [2] The commit message format of geode. <https://cwiki.apache.org/confluence/display/GEODE/Commit+Message+Format>, 2020.
- [3] M. Allamanis, H. Peng, and C. Sutton. A convolutional attention network for extreme summarization of source code. In *Proc. ICML*, pages 2091–2100, 2016.
- [4] R. Buse and W. Weimer. Automatic documentation inference for exceptions. In *Proc. ISSA*, pages 273–282, 2008.
- [5] Q. Chen and M. Zhou. A neural framework for retrieval and summarization of source code. In *Proc. ASE*, pages 826–831, 2018.
- [6] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin. Deep code comment generation. In *Proc. ICPC*, pages 200–210, 2018.
- [7] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin. Deep code comment generation with hybrid lexical and syntactical information. *Empirical Software Engineering*, pages 1–39, 2019.
- [8] X. Hu, G. Li, X. Xia, D. Lo, S. Lu, and Z. Jin. Summarizing source code with transferred api knowledge. In *Proc. IJCAI*, pages 2269–2275, 2018.
- [9] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer. Summarizing source code using a neural attention model. In *Proc. ACL*, pages 2073–2083, 2016.
- [10] Z. M. Jiang and A. E. Hassan. Examining the evolution of code comments in postgresql. In *Proc. MSR*, pages 179–180, 2006.
- [11] X. Li, H. Jiang, D. Liu, Z. Ren, and G. Li. Unsupervised deep bug report summarization. In *Proc. ICPC*, pages 144–154, 2018.
- [12] Z. Li and H. Zhong. An empirical study on obsolete issue reports. In *Proc. ASE*, page to appear, 2021.
- [13] M. Linares-Vásquez, L. F. Cortés-Coy, J. Aponte, and D. Poshyvanyk. Changescribe: A tool for automatically generating commit messages. In *Proc. ICSE*, pages 709–712, 2015.
- [14] L. Moreno, J. Aponte, G. Sridhara, A. Marcus, L. Pollock, and K. Vijay-Shanker. Automatic generation of natural language summaries for java classes. In *Proc. ICPC*, pages 23–32, 2013.
- [15] S. Rastkar, G. C. Murphy, and G. Murray. Automatic summarization of bug reports. *IEEE Transactions on Software Engineering*, 40(4):366–380, 2014.
- [16] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker. Towards automatically generating summary comments for Java methods. In *Proc. ASE*, pages 43–52, 2010.
- [17] I. Sutskever, O. Vinyals, and Q. V. Le. Sequence to sequence learning with neural networks. In *Proc. NIPS*, pages 3104–3112, 2014.
- [18] 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.
- [19] E. Wong, T. Liu, and L. Tan. Clocom: Mining existing source code for automatic comment generation. In *Proc. SANER*, pages 380–389, 2015.
- [20] E. Wong, J. Yang, and L. Tan. Autocomment: Mining question and answer sites for automatic comment generation. In *Proc. ASE*, pages 562–567, 2013.
- [21] S. Xu, Y. Yao, F. Xu, T. Gu, H. Tong, and J. Lu. Commit message generation for source code changes. In *IJCAI*, pages 3975–3981, 2019.
- [22] S. Zhang, C. Zhang, and M. D. Ernst. Automated documentation inference to explain failed tests. In *Proc. ASE*, pages 63–72, 2011.