Towards Refactoring-Aware Code Review

Xi Ge Saurabh Sarkar Emerson Murphy-Hill
Department of Computer Science
NC State University
Raleigh, NC, USA 27695
{xge, ssarkar4}@ncsu.edu, emerson@csc.ncsu.edu

ABSTRACT

Software developers review changes to a code base to prevent new bugs from being introduced. However, some parts of a change are more likely to introduce bugs than others, and thus deserve more care in reviewing. In this short paper, we discuss our ongoing work to build a reviewing tool that automatically determines which changes in a change set are refactorings, uses this information to help the developer distinguish between refactoring and non-refactoring changes, and ultimately reduces the time it takes developers to review code accurately. We also discuss the challenges and opportunities we have faced when building this refactoring-aware code review tool.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Code inspections and walkthroughs*

General Terms

Design, Human Factors

Keywords

Code Review, Refactoring, Refactoring Detection

1. INTRODUCTION

Code review is a common practice in both open source and industrial software projects [1]. In a code review, a developer inspects modifications, such as bug fixes or feature enhancements, to an existing piece of code. There are many reasons to review code, but the main reason is to find bugs [1]. One common part of a code change is a *refactoring* [9, 11], which includes renaming variables and breaking large methods into smaller ones. Refactorings, by definition, maintain the behavior of a program, and so cannot introduce bugs. Nonetheless, new bugs may be introduced by change sets containing both refactorings and non-refactorings; our previous work indicates that such change sets are common [9].

CHASE '14, June 2 – June 3, 2014, Hyderabad, India

Because correctly-performed refactorings must not contain bugs, developers do not need to focus as much effort on the refactorings in a change set. In fact, reviewing refactorings as rigorously as non-refactorings is undesirable — reviewing refactorings distracts developers from non-refactorings, which may contain bugs. There is also a more subtle reason why spending effort on reviewing refactorings is undesirable. The reason is that if a refactoring is done with an automated refactoring tool, the tool validates the correctness of the refactoring, yet evidence of that validation is not carried through to the code review. Likewise, if a refactoring is validated manually by a developer for correctness, that validation is likewise missing during code review. Thus, re-validation of such refactorings are a duplication of effort.

To illustrate the problem, Figure 1 shows part of a change set that three developers reviewed and illustrates. Notice that this existing review tool does not help developers differentiate between refactorings and non-refactorings. The body of the `if` statement is a moderately complex refactoring, yet the change to the conditional, a new expression examining `fComparefilters`, Thus, the tool does not help the developer focus on the potentially-buggy part of the change.

2. RELATED WORK

Tools for reviewing code changes include Mondrian,[1] Gerrit,[2] and CodeFlow [1]. Some make it easier for developers to review code motion (e.g., moved files), but are, by design, language-agnostic, precluding them from analyzing refactorings. Several tools can detect refactorings, including Refactoring Crawler [2], Change Distiller [5], and Ref-Finder [10]. These approaches support reverse engineering, rather than code review. As a result, they tend to be tolerant of false positives (identifying a change as a refactoring when it actually changes behavior), whereas for refactoring-aware code review tools, false positives must be minimized. Kim and colleagues specifically suggested, based on their study at Microsoft, that these refactoring detection approaches should be applied to refactoring-aware code review [7].

3. APPROACH

We are developing a tool that distinguishes refactorings from non-refactorings during code review, investigating three areas in the process.

[1] `https://developers.google.com/appengine/articles/rietveld`
[2] `https://code.google.com/p/gerrit/`

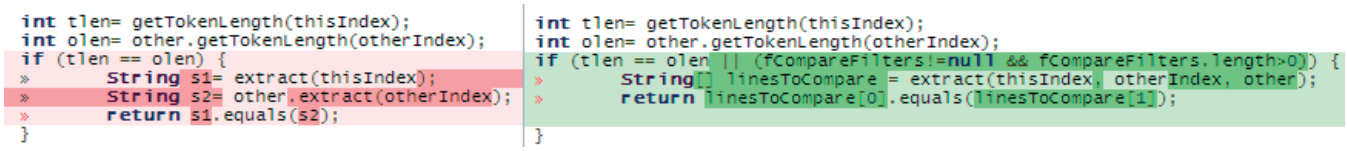


Figure 1: A change shown in the Gerrit code review tool. (http://goo.gl/nMCSL)

The first area we are investigating capitalizes on the record kept by some refactoring tools, of the refactorings applied to a codebase.[3] We leverage this record by passing it on to the code review tool, which can then identify which parts of the code change were part of refactorings. Code reviews supported by this approach will be easier when there is a record of tool-supported refactorings available. As a side effect, this approach may encourage developers to use refactoring tools by amplifying their benefits.

The second area we are investigating detects refactorings that are performed manually, since our research suggests that 9 out of 10 refactorings are done without tools [9]. We leverage the results of our previous work where we identify refactorings that a developer is performing manually to offer her a way to finish the refactoring automatically [6]. By extending this previous work, we can detect refactorings without the need to monitor the developer's development environment. Our prior work suggests that, while not every refactoring can be identified with 100% accuracy, common refactorings (like *rename*) can be statically detected by identifying prototypical micro-changes, then validating whether those micro-changes constitute a complete and correct refactoring.

The third area we are investigating is how to differentiate refactorings from non-refactorings in the code review tool's user interface. Our simple approach is to simply not show refactorings, in the same way that some review tools ignore changes to whitespace. The research challenge is to present this information in an developer-friendly way.

4. IMPLEMENTATION

We have implemented a prototype of our approach in the Eclipse development environment, using existing refactoring histories and a basic user interface. Figure 2a displays the standard Eclipse difference tool, which can be used for code review. We have augmented this standard difference tool to include a context menu item that says "ignore refactorings". Once clicked, refactorings are hidden (Figure 2b).

This prototype tool works by taking two versions of an Eclipse project, call them A^{old} and A^{new}, each consisting a set of source files. This pair may come from any source, including arbitrary versions of a project under version control or a single project with an accompanying patch. The tool works as follows:

1. The tool looks for `.history` files in A^{new}, which contain a record of which refactorings were performed on the project, and selects only those refactorings that are new since A^{old}.

2. The tool replays those refactorings on A^{old}, creating a new version, $A^{old+refactoring}$.

3. The tool differences $A^{old+refactoring}$ and A^{new} to produce diff $^{non\text{-}refactoring}$.

4. The tool displays A^{old} on the left side of the editor, A^{new} on the right side, and finally displays the diff $^{non\text{-}refactoring}$ over the top of the two.

Using this process, only non-refactorings are visually highlighted in the editor, thus allowing developers to focus just on the potentially buggy parts of the change.

5. DISCUSSION

Our implementation efforts thus far have exposed several interesting challenges and opportunities for code review tools. Let us first discuss challenges and opportunities from a program analysis viewpoint, and then from a user interface viewpoint.

5.1 Program Analysis

One of the challenges that we have seen so far is how to deal with changes where both refactorings and non-refactorings are made in close proximity. For example, suppose a developer starts with this code (v1):

```
System.out.println("foo");
```

Then changes it to this:

```
System.out.println(m());
```

Then finally refactors it to this (v2):

```
String s = m();
System.out.println(s);
```

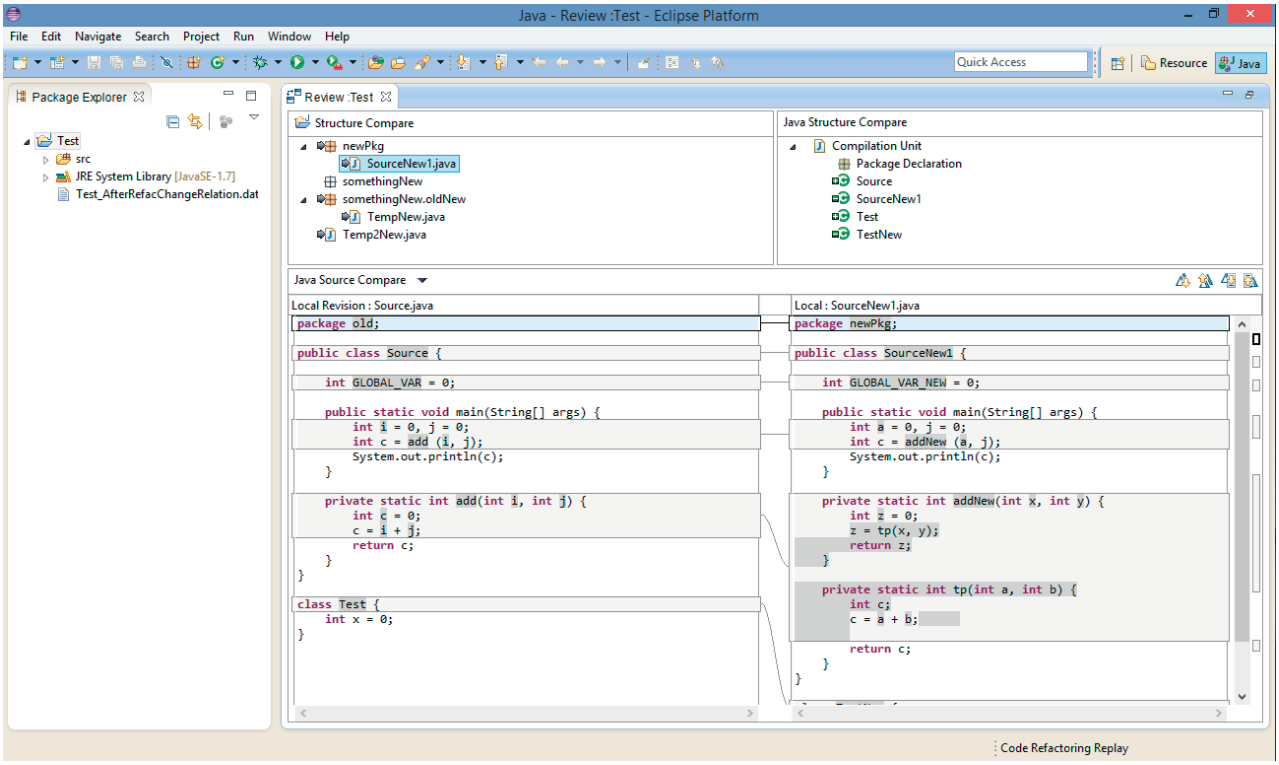
Now Suppose you wanted to review the change, between v1 and v2. Using our current tool, you would not be able to replay the refactoring, because the Extract Local Variable refactoring was applied to an intermediate version of the program (containing `m()`) that was not visible in v1. What our tool currently does is mark the totality of the change as a non-refactoring – this conservative approach respects our main design criteria, that there should be no false positives. However, a more sophisticated version of the tool could probably more cleanly divide the refactoring from the non-refactoring, and still obey our design criteria.

A second challenge is a more fundamental question about the meaning of refactoring. Suppose a developer starts with this code (v1):

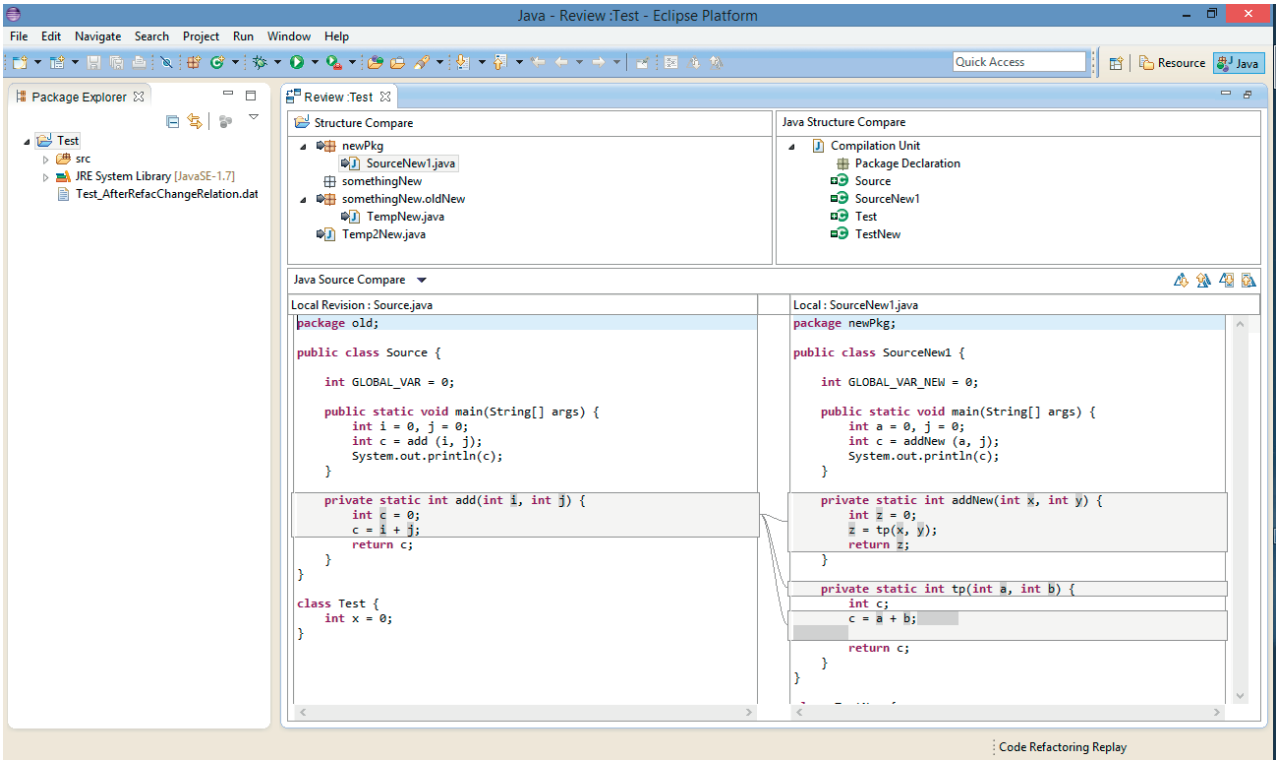
```
int x;

void setX(int newX){
   x = newX;
}
```

[3] `http://goo.gl/42Zyy`



(a) Eclipse's basic code review tool, highlighting the differences between some original toy code (at left) and the code after some changes (at right)



(b) Our prototype building on Eclipse's code review tool, with refactorings hidden.

Figure 2: The standard Eclipse comparison and our tool.

```
void foo(int y){
   System.out.println(x);
}
```

The developer then makes some changes and ends with this code:

```
int z;

void setX(int newX){
   z = newX;
}

void foo(int y){
   System.out.println(y);
}
```

In this case, we can infer that `x` was renamed to `z`, yet at the same time, the last reference `x` was changed to reference `y` instead. Was a refactoring performed here? If so, what parts should be hidden, and what parts should be emphasized? Thus far, we are unsure about the answers to these questions.

5.2 User Interface

Apart from program analysis challenges, user interfaces challenges also remain. For example, is eliding refactorings the best way to de-emphasize them? While theoretically refactorings do not change program behavior, practically speaking, developers may still want to review refactoring changes. For example, developers may not trust refactoring tools, or may want to review refactorings to learn from their peers or to check adherence to architectural styles. We are considering alternatives to hiding refactorings, such as by using a different highlighting color or by using the same color, just a lighter tone.

In a broader sense, this code review research can be thought of at a more abstract level than refactorings. How can we focus a developers' attention during code review on *the parts of the program that matter?* In this research, we have assumed that refactorings are less important than other types of changes, but gradations of importance might be mapped to other features as well. For instance, other less important code entities for developers to review might include comments. Conversely, more important code entities for review might be those marked as buggy by a static analysis tool [3] or those marked as defect-prone via historical analysis [4]. Bringing such information to the developers' attention (or drawing her attention away) may be a useful approach to improving the effectiveness of code review.

Instead of emphasizing or deemphasizing program entities, a user interface that aids in code review could help developers navigate a change set in a more logical way. For example, consider that present day code review is largely done as on a line-by-line or file-by-file basis, even though conceptually related parts of changes may be scattered throughout a system [8]. Thus, reviewing changes using traditional code review tools may be a cognitively demanding task. One way to make this task easier would be to present program entities both in conceptual logical groupings, and perhaps also presents groups in an order that minimizes the cognitive burden. From a refactoring perspective, we interpret this to mean that refactorings should be reviewed separately from non-refactorings, but perhaps also that refactorings of different types should be reviewed separately. This has implications beyond refactorings as well; for instance, a future code review tool might present changes related to a database, then changes related to logging, then changes to comments, and finally any remaining changes.

6. CONCLUSION

In this paper, we have presented the beginnings of a tool that aims to ease code review by de-emphasizing refactorings that do not change program behavior. We plan on performing a study to determine how and when this approach is useful for code review. In a larger sense, this investigation into refactoring-aware code review has suggested several opportunities for easing code review in other ways.

7. ACKNOWLEDGMENTS

Thanks to our reviewers for their insightful comments. Thanks to Google for funding this research.

8. REFERENCES

[1] A. Bacchelli and C. Bird. Expectations, outcomes, and challenges of modern code review. In *Proceedings of the 35th International Conference on Software Engineering*, 2013.

[2] D. Dig, C. Comertoglu, D. Marinov, and R. Johnson. Automated detection of refactorings in evolving components. In *Proceedings of ECOOP*, pages 404–428, 2006.

[3] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of the Symposium on Operating System Design & Implementation*. USENIX, 2000.

[4] N. E. Fenton and N. Ohlsson. Quantitative analysis of faults and failures in a complex software system. *Software Engineering, IEEE Transactions on*, 26(8):797–814, 2000.

[5] B. Fluri, M. Wursch, M. Pinzger, and H. Gall. Change distilling:tree differencing for fine-grained source code change extraction. *Software Engineering, IEEE Transactions on*, 33(11):725–743, 2007.

[6] X. Ge, Q. L. DuBose, and E. Murphy-Hill. Reconciling manual and automatic refactoring. In *Proceedings of the 2012 International Conference on Software Engineering*, pages 211–221, 2012.

[7] M. Kim, T. Zimmermann, and N. Nagappan. A field study of refactoring challenges and benefits. In *Proceedings of FSE '12*, pages 1–11, 2012.

[8] S. Letovsky and E. Soloway. Delocalized plans and program comprehension. *Software, IEEE*, 3(3):41–49, 1986.

[9] E. Murphy-Hill, C. Parnin, and A. P. Black. How we refactor, and how we know it. *Software Engineering, IEEE Transactions on*, 38(1):5–18, 2012.

[10] K. Prete, N. Rachatasumrit, N. Sudan, and M. Kim. Template-based reconstruction of complex refactorings. In *Software Maintenance (ICSM), 2010 IEEE International Conference on*, pages 1–10, 2010.

[11] Z. Xing and E. Stroulia. Refactoring practice: How it is and how it should be supported-an Eclipse case study. In *International Conference on Software Maintenance*, pages 458–468. IEEE, 2006.