RefDistiller: A Refactoring Aware Code Review Tool for Inspecting Manual Refactoring Edits

Everton L. G. Alves[†‡] Myoungkyu Song[†] Miryung Kim[§]
University of Texas at Austin, USA[†] University of California, Los Angeles, USA[§]
Federal University of Campina Grande, Brazil[‡]
{everton, mksong1117}@utexas.edu, miryung@cs.ucla.edu

ABSTRACT

Manual refactoring edits are error prone, as refactoring requires developers to coordinate related transformations and understand the complex inter-relationship between affected types, methods, and variables. We present RefDistiller, a refactoring-aware code review tool that can help developers detect potential behavioral changes in manual refactoring edits. It first detects the types and locations of refactoring edits by comparing two program versions. Based on the reconstructed refactoring information, it then detects potential anomalies in refactoring edits using two techniques: (1) a template-based checker for detecting *missing edits* and (2) a refactoring separator for detecting *extra edits* that may change a program's behavior. By helping developers be aware of deviations from pure refactoring edits, RefDistiller can help developers have high confidence about the correctness of manual refactoring edits. RefDistiller is available as an Eclipse plug-in at `https://sites.google.com/site/refdistiller/` and its demonstration video is available at `http://youtu.be/0Iseoc5HRpU`.

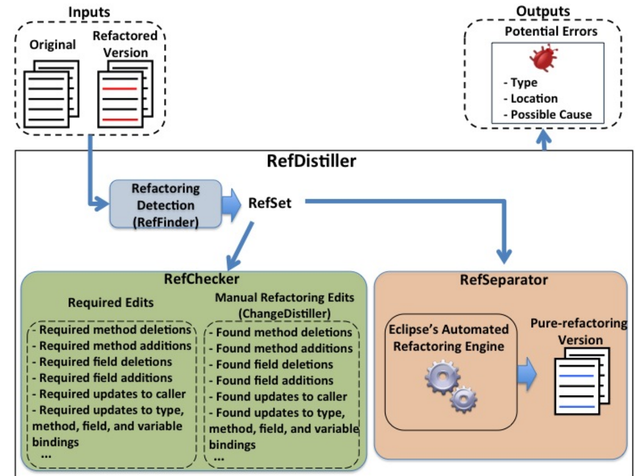

Figure 1: Overview of RefDistiller

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: [Distribution, Maintenance, and Enhancement]

General Terms

Design, Experimentation

Keywords

Software evolution, refactoring

1. INTRODUCTION

Recent studies show that developers often conduct refactorings manually despite their awareness of automated refactoring engines [5, 7, 12]. Most expert developers do refactoring edits manually instead of using automated refactoring tools [7]. Developers also underuse and misuse automated refactoring tools [12].

Manual refactorings are error prone. According to a field study at Microsoft [5], 77% of developers find it hard to perform manual refactorings correctly. Weißgerber and Diehl find evidence that bugs are caused by incomplete refactorings [13]. Existing approaches for detecting manual refactoring anomalies are limited. Rachatasumrit and Kim find that regression testing suites in practice are often inadequate for covering refactored locations and are ineffective in detecting refactoring errors [10]. SafeRefactor [11] validates refactoring edits by leveraging an existing test generation engine and by comparing test results between the old and new program versions. However, it also requires having enough test coverage. GhostFactor [4] detects a limited category of missing edits in manual refactoring but does not detect extra edits that may change a program's behavior.

In this tool demonstration paper, we present RefDistiller, a refactoring-aware code review Eclipse plug-in. Figure 1 shows the overview of RefDistiller. To detect potential deviations from pure refactoring edits, RefDistiller incorporates two key techniques: (1) RefChecker for detecting missing edits and (2) RefSeparator for detecting extra edits. It takes an original version and a manual refactored version as input, and automatically infers the types and locations of potential refactoring edits using RefFinder [9]. For each refactoring edit, RefChecker checks a set of required

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

FSE'14, November 16–21, 2014, Hong Kong, China
Copyright 2014 ACM 978-1-4503-3056-5/14/11...$15.00
http://dx.doi.org/10.1145/2635868.2661674

code modifications and checks the preservation of method or field reference bindings using predefined template rules. If the required changes are not reflected in the actual edits, those missing edits are reported to the user, together with information on their location, type, and problematic reference bindings. Similarly, for each refactoring edit, RefSeparator automatically applies an equivalent pure refactoring using a modified version of the Eclipse refactoring engine to create a pure refactoring version. This version is then compared against the manual refactoring version using ChangeDistiller's syntactic differencing technique [3]. If the versions are not identical, RefSeparator reports the locations of extra edits.

Currently, RefDistiller supports six of most common refactoring types for Java programs: `move method`, `extract method`, `inline method`, `rename method`, `pull up method`, and `push down method`. This demo paper's main contribution is to describe different views of RefDistiller tool and to provide details on how users can use it in the context of inspecting manual refactoring edits using a motivating scenario. The algorithm and evaluation of RefDistiller are detailed in our technical report [1]. The tool is available at the following web site `https://sites.google.com/site/refdistiller/`.

2. MOTIVATING SCENARIO AND TOOL FEATURES

Suppose Ann performs two refactoring edits manually, a `pull up method` refactoring and an `extract method` refactoring. Figure 3 shows the old and new version of manual refactoring edits.

Ann moves method `getPrice` from class `EBookManager` to its superclass `BookManager` but she is unaware that she mistakenly has overridden an existing method `SupplyManager.getPrice` (see ① in Figure 3). Because there are no compilation errors, Ann does not suspect that she changed the program behavior accidentally. For example, after the update, `BookManager.rent` calls `BookManager.getPrice` instead of `SupplyManager.getPrice` (see ③ and ④ in Figure 3). In this case, RefDistiller detects a missing edit in line 17 of `BookManager.java` and reports a warning message, 'Detected a problematic reference binding which may have affected method rent'. By examining the warning, Ann recognizes that she should have updated line 17 so that it calls `super.getPrice(book, days)` instead of `getPrice(book, days)`.

Ann performs an extract method refactoring in `BookManager.findBook` by creating a new method `checkTitle` which encapsulates `book.getTitle().equals(title)` at line 8 (see ② in Figure 3). While carrying out this extract method refactoring, Ann adds a call to `setUnavailable(book)` at line 9.

During peer code review, Kate, Ann's manager, sees this warning message generated by RefDistiller, 'Detected extra edits that may have changed the behavior of method p1.BookManager.findBook' at line 9. She then double-checks with Ann whether this deviation from pure refactoring edits is a correct edit intended by Ann.

Input Selection. Kate opens the main view by selecting a menu option of Window → Show View → Other →RefDistiller. Kate uses a drop-down menu to specify the original version before Ann's manual refactoring edits and the target version after her edits (see the menu highlighted in blue in Figure 2).

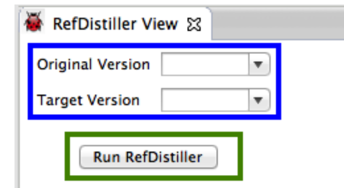

Figure 2: RefDistiller input view.

RefDistiller Results View. When Kate presses the Run RefDistiller button in Figure 2, it first runs RefFinder [9] to detect the refactoring edits performed and their locations. RefFinder uses a logic query based approach for reconstructing common types of refactoring edits from two program versions. RefDistiller then runs the RefChecker and RefSeparator modules, searching for missing edits and extra edits that deviate from pure refactoring edits.

The first tab, **Potential Problems** shows all potential deviations from pure refactoring edits reported by both RefChecker and RefSeparator. Each line describes the type of a refactoring anomaly such as `BINDING_PROBLEM`, the location (e.g. method `rent` from class `p1.BookManager`), and a short description of the problem. By clicking each line, Kate now reviews the corresponding Java files, where the potential problems are located.

Missing Edits View. By clicking the **Missing Edits** tab, Kate sees the problems reported from RefChecker. RefChecker currently uses data flow analysis and type binding resolution analysis to check required method and field deletions (or additions), required updates to callers of the refactored methods, and required updates to type, method, field, and variable bindings.

RefChecker reports the following 9 types of warning messages. The details on our rule-based checking algorithm are described in our technical report [1].

- **Missing method:** Method X was not found in Class Y.
- **Not removed method:** Method X should have been removed from Class Y.
- **Missing statement update:** There is at least one missing statement to be updated in the method X's body.
- **Missing statement addition:** There is at least one missing statement to be inserted in the method X's body.
- **Missing statement deletion:** There is at least one missing statement to be deleted in the method X's body.
- **Missing type update:** The type associated with field X needs to be updated.
- **Binding problem:** Detected problematic reference bindings which may have affected method X.
- **Visibility problem:** Method X is not visible for one of its callers.
- **Missing renaming:** Method X was not renamed.

By clicking each line, the respective Java file is opened and a warning message is tagged at the exact location of the warning (see Figure 3). Kate can now see that that Ann should have updated line 17 by writing `super.getPrice(book, days)` instead of `getPrice(book, days)`, because `rent` used to call `SupplyManager.getPrice` in the old version but now calls `BookManager.getPrice` in the new version instead.

Extra Edits View. By clicking the **Extra Edits** tab, Kate can see the problems reported from RefSeparator. By clicking each line, she can open the **Eclipse Compare View** and examine the program difference between a manual refac-

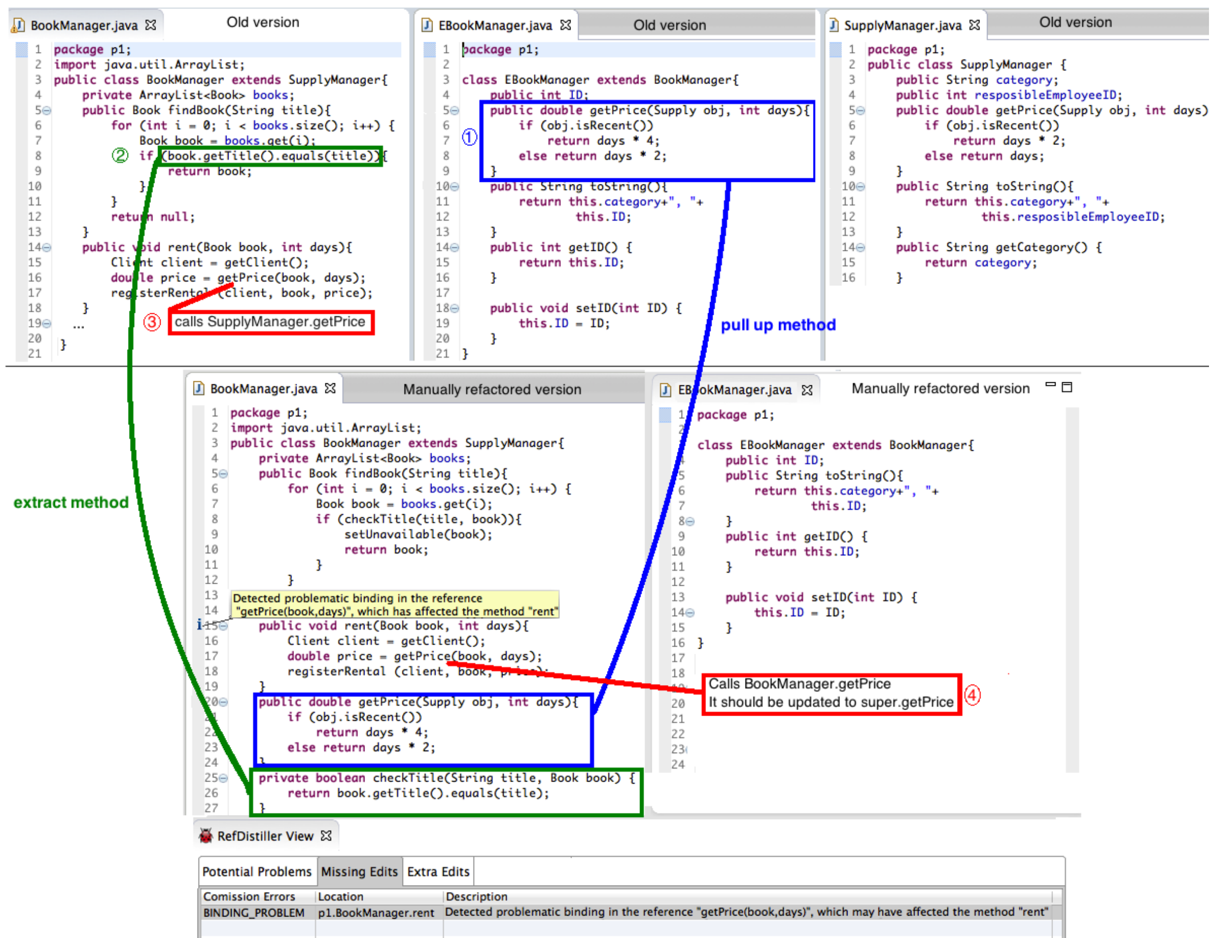

Figure 3: Ann moved `getPrice` from `EBookManager` to `BookManager`, mistakenly overriding `SupplyManager.getPrice`. Thus, `BookManager.rent` should have been updated to call `super.getPrice(book, days)` instead to ensure behavior preservation.

toring version and a pure refactoring version generated by Eclipse's automated refactoring API. Figure 4 shows a pure refactoring version on the left side and Ann's manual version on the right side. When the two versions are different, RefSeparator reports that the manual version is not pure refactoring and pinpoints the location of extra edits, in this case line 9 calling `setUnavailable`. When Kate sees a warning message, 'Detected extra edits that may change the behavior of method findBook', she can check with Ann whether she intended to introduce this new behavior.

3. RELATED WORK

Formal verification is an alternative for avoiding refactoring anomalies. Cornélio et al. [2] propose rules for guaranteeing semantic preservation. Mens et al. [6] use graph rewriting for specifying refactorings. Overbey et al. [8] present a collection of refactoring specifications for Fortran 95. However, these approaches focus on improving the correctness of automated refactoring through formal specifications, as opposed to finding refactoring anomalies during manual edits.

The closest work to ours is GhostFactor [4]. GhostFactor automatically checks correctness of manually performed refactoring by checking required conditions. The idea of GhostFactor is similar to RefChecker's templates; however, it handles three refactoring types. Currently, RefChecker reports nine types of warnings for six refactoring types. In contrast to RefSeparator, GhostFactor cannot detect extra edits from manual refactorings.

4. SUMMARY

Recent studies show that developers often mix refactoring with other semantic changes and developers do most refactoring manually. We present the features of RefDistiller, a refactoring-aware code review tool that can help developers detect potential errors in manual refactoring edits. It points out the locations where incomplete refactorings are applied and code elements that diverge from a pure refactoring modification. While extra edits performed during refactoring are not always errors and could be intentionally made, we believe that pinpointing these extra edits could help developers focus their attention to behavior changes during peer code reviews. Developers may leverage RefDistiller to gain high confidence that their refactoring edits preserve the already established behavior, to localize and fix refactoring bugs, or

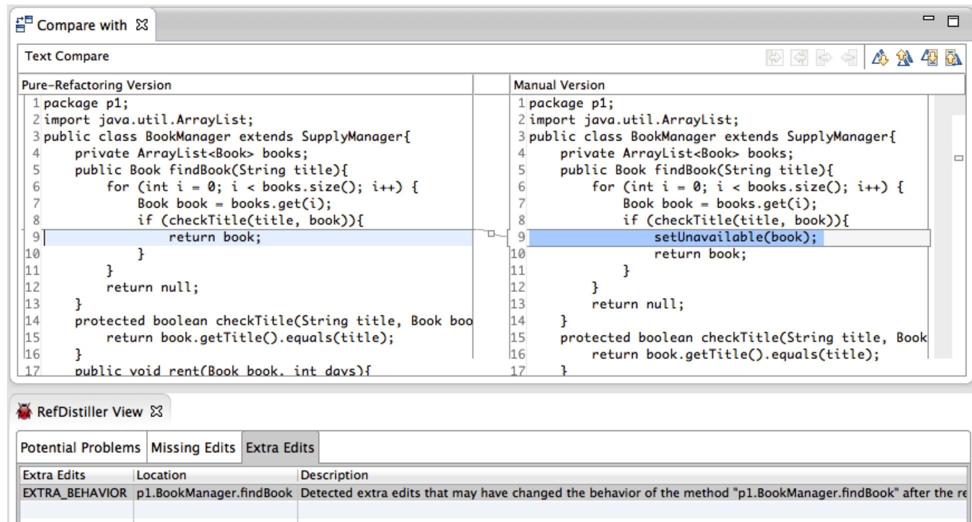

Figure 4: Ann created a new method `checkTitle` by extracting a condition (line 8 - Figure 3) but also added a new method call to `setUnavailable`.

to check whether extra edits are correct intended edits.

In the evaluation detailed in our separate technical report [1], we evaluate RefDistiller's effectiveness on a data set with one hundred manual refactoring bugs. These bugs are hard to detect because they do not produce any compilation errors nor are caught by the pre- and post-condition checking of many existing refactoring engines. RefDistiller can identify 97% of the erroneous edits, of which 24% are not detected by extensive, automatically generated test suites.

5. ACKNOWLEDGMENT

This work was supported by National Science Foundation under grants CCF-1149391, CCF-1117902, SHF-0910818, CNS-1239498, a Google Faculty Award, and by National Institute of Science and Technology for Software Engineering, funded by CNPq/Brasil, grant 573964/2008-4.

6. REFERENCES

[1] E. L. Alves, M. Song, M. Kim, P. D. Machado, and T. Massoni. Refdistiller: Detecting anomalies in manual refactoring edits. Technical report, University of Texas at Austin, TR-ECE-2014-3, March, 2014.

[2] M. Cornélio, A. Cavalcanti, and A. Sampaio. Sound refactorings. *Science of Computer Programming*, 75(3):106–133, 2010.

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

[4] X. Ge and E. Murphy-Hill. Manual refactoring changes with automated refactoring validation. In *Proc. of ICSE'14*. IEEE, 2014.

[5] M. Kim, T. Zimmermann, and N. Nagappan. A field study of refactoring challenges and benefits. In *Proc. of FSE'12*, page 50. ACM, 2012.

[6] T. Mens, N. Van Eetvelde, S. Demeyer, and D. Janssens. Formalizing refactorings with graph transformations. *Journal of Software Maintenance and Evolution: Research and Practice*, 17(4):247–276, 2005.

[7] S. Negara, N. Chen, M. Vakilian, R. E. Johnson, and D. Dig. A comparative study of manual and automated refactorings. In *Proc. of ECOOP'13*, pages 552–576. Springer, 2013.

[8] J. L. Overbey, M. J. Fotzler, A. J. Kasza, and R. E. Johnson. A collection of refactoring specifications for fortran 95. In *ACM SIGPLAN Fortran Forum*, volume 29, pages 11–25. ACM, 2010.

[9] K. Prete, N. Rachatasumrit, N. Sudan, and M. Kim. Template-based reconstruction of complex refactorings. In *Proc. of ICSM'10*, pages 1–10. IEEE, 2010.

[10] N. Rachatasumrit and M. Kim. An empirical investigation into the impact of refactoring on regression testing. In *Proc. of ICSM'12*, pages 357–366. IEEE, 2012.

[11] G. Soares, R. Gheyi, D. Serey, and T. Massoni. Making program refactoring safer. *Software, IEEE*, 27(4):52–57, 2010.

[12] M. Vakilian, N. Chen, S. Negara, B. A. Rajkumar, B. P. Bailey, and R. E. Johnson. Use, disuse, and misuse of automated refactorings. In *Proc. of ICSE'12*, pages 233–243. IEEE, 2012.

[13] P. Weißgerber and S. Diehl. Are refactorings less error-prone than other changes? In *Proc. of MSR'06*, pages 112–118. ACM, 2006.