A Tool Supporting Postponable Refactoring

Katsuhisa Maruyama
Department of Computer Science
Ritsumeikan University, Japan
maru@cs.ritsumei.ac.jp

Shinpei Hayashi
Department of Computer Science
Tokyo Institute of Technology, Japan
hayashi@se.cs.titech.ac.jp

***Abstract*—Failures of precondition checking when attempting to apply automated refactorings often discourage programmers from attempting to use these refactorings in the future. To alleviate this situation, the postponement of the failed refactoring instead its cancellation is beneficial. This poster paper proposes a new concept of postponable refactoring and a prototype tool that implements postponable EXTRACT METHOD as an Eclipse plug-in. We believe that this refactoring tool inspires a new field of reconciliation automated and manual refactoring.**

***Keywords*-software evolution; software refactoring; precondition checking; code change management**

I. INTRODUCTION

Refactoring improves the design of existing code without changing its behavior [1]. Therefore, modern Integrated Development Environments (IDEs), including Eclipse, IntelliJ IDEA, and Visual Studio support several automated refactorings that enable programmers to easily apply the behavior-preserving transformations of existing code. Nonetheless, they greatly underuse automated refactoring tools [2]–[6]. This is mostly due to usability problems and thus several improvements of existing refactoring tools have been proposed. Especially, WitchDoctor [7], BeneFactor [8], and GhostFactor [9] emphasize the significance of reconciling manual and automated refactorings to make refactoring tools more usable.

Among the usability problems, precondition checking [10] might be an obstacle to the widespread use of automated refactorings, which is adopted in almost all refactoring tools. In general, the preconditions should be strong enough to prevent the applied refactoring from breaking compilable code or altering its behavior. On the other hand, error messages resulting from precondition checking frequently discourage programmers from activating the applied refactoring again [11]. From this perspective, several studies have revealed overly strong (pre)conditions in refactoring [12], [13]. We believe that precondition checking remains in every refactoring tool although some of the preconditions are overly strong and many programmers prefer to relax them, since precondition failures are foreboding signs. Programmers should keep in mind that the disregard of such failures might involve troublesome manual review of the behavior of the changed code.

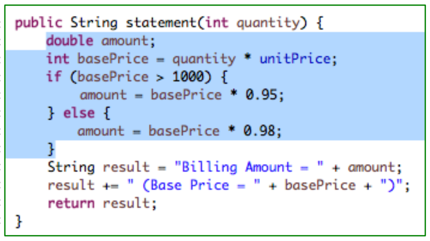
This poster paper presents a new kind of refactoring tool that can postpone the application of automated refactorings to alleviate one aspect of the usability problems. The tool allows a programmer to suspend the execution of the applied refactoring if its preconditions are not satisfied and to restart the suspended refactoring once all the preconditions are satisfied. This postponement brings a situation in which she can freely change code by hand or by automated transformations (e.g., refactoring and quick assist) while checking whether the suspended refactoring can be restarted. Restarting never requires her to once again identify the target code fragments and to reconfigure the refactoring settings of options. The refactoring contexts (the identified code and the configuration settings) provided when she activated a refactoring before is automatically stored in the tool. The tool manages them depending on the later code changes, and also recalls them so that the tool can restart the suspended refactoring without her unnecessary intervention. The prototype tool currently supports the postponement of EXTRACT METHOD, which is implemented as an Eclipse plug-in.

II. POSTPONABLE REFACTORING TOOL

The Eclipse refactoring tool checks several predefined preconditions and reports its result with a severity level of information, warning, error, or fatal error. Errors predict the introduction of compilation errors and non-behavior preservation changes. The (normal) errors advise a programmer to abort the applied refactoring, and the fatal errors quit continuing the execution of the refactoring.

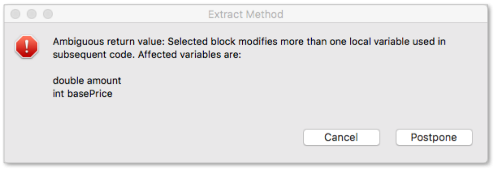
In postponable refactoring, some of the normal and fatal errors are reclassified into recoverable ones. As a result, it treats three kinds of errors: normal, fatal, and recoverable. Theoretically, all the errors seem to be recoverable if the identified code fragments are completely rewritten. However, it might be hard for programmers to maintain the refactoring contexts of a suspended refactoring during a large rewrite of the code. Thus, in the current prototype tool, we addressed two precondition failures of the EXTRACT METHOD: “*Ambiguous return value*” as a fatal error and “*Already existing method name*” as a normal error, which are both easy to recover. In the EXTRACT METHOD, the modified values of local variables within the newly extracted method must be returned from it if those values are accessed after the invocation of the extracted method. Moreover, the name of the extracted method should differ from the names already existing in the class defining the extracted method to avoid a compilation error resulting from name duplication.

For example, in Figure 1, when the EXTRACT METHOD is applied to the code fragments within lines 2–8, an “*Ambiguous return value*” error occurs for variables `amount` and



```
public String statement(int quantity) {
    double amount;
    int basePrice = quantity * unitPrice;
    if (basePrice > 1000) {
        amount = basePrice * 0.95;
    } else {
        amount = basePrice * 0.98;
    }
    String result = "Billing Amount = " + amount;
    result += " (Base Price = " + basePrice + ")";
    return result;
}
```

Fig. 1. Code snippet that is a target of EXTRACT METHOD.



Extract Method

Ambiguous return value: Selected block modifies more than one local variable used in subsequent code. Affected variables are:

double amount
int basePrice

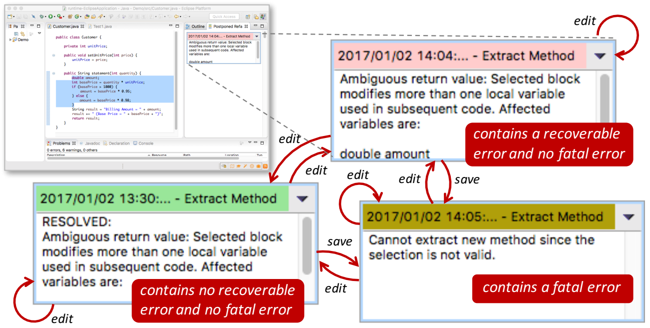
Fig. 2. Fatal error report of postponable EXTRACT METHOD.

`basePrice`. If a programmer uses the Eclipse's original refactoring tool, she must cancel EXTRACT METHOD and edit the problematic code (INLINE TEMP or REPLACE TEMP WITH QUERY for `basePrice` might be feasible). After that, she will restart the EXTRACT METHOD from the beginning by identifying code fragments and selecting the menu item. The burden of this restarting might lead her to manually refactor the code.

Differing from the original automated refactoring, a postponable refactoring encourages a programmer to continue automatic application of refactoring. Figure 2 depicts a fatal error dialog of the postponable EXTRACT METHOD, appearing when the precondition checking detects a recoverable error and no fatal error. She can suspend the execution of this refactoring by clicking the 'Postpone' button that is added to the dialog. The suspended refactoring appears in a view located on the right side of the editor as shown in Figure 3. The red bar represents that the current status is under temporary suspension.

The postponable refactoring tool monitors any editing of code fragments specific to a suspended refactoring. In this example, it monitors the whole body of method `statement()` including the code fragments a programmer identified at the beginning. It also keeps track of all editing within the files related to the monitored code fragments by employing ChangeMacroRecorder [14] and also adjusts the selection range. It records every editing action that a programmer performed on the Eclipse Java editor in the background, which is the same as OperationRecorder [15], Fluorite [16], and CodingTracker [17].

Each time the postponable refactoring tool detects an edit of the monitored code fragments, it attempts to check preconditions of the suspended refactoring. If the precondition checking reports no recoverable error and no fatal error, the current status is shifted to restartable in which the bar is colored with green. Otherwise, the current status is not changed. One more trigger of the status transition is a file-save action. When a programmer saves the file that contains the monitored code fragments, the tool checks the preconditions of the suspended



2017/01/02 14:04:... - Extract Method

Ambiguous return value: Selected block modifies more than one local variable used in subsequent code. Affected variables are:

double amount

2017/01/02 13:30:... - Extract Method

RESOLVED:
Ambiguous return value: Selected block modifies more than one local variable used in subsequent code. Affected variables are:

2017/01/02 14:05:... - Extract Method

Cannot extract new method since the selection is not valid.

edit
contains a recoverable error and no fatal error
edit
edit
edit
edit
save
contains no recoverable error and no fatal error
save
edit
contains a fatal error
edit

Fig. 3. Status of postponable EXTRACT METHOD.

refactoring. If the precondition checking reports a fatal error, the color of the bar becomes brown.

The current status of each suspended refactoring is determined by the worst severity level of precondition failures and changed as shown in Figure 3. Whenever the bar is colored with green, a programmer can restart the suspended refactoring. She can also cancel it anytime if it does not need to be applied hereafter. The canceled refactoring is removed from the view. The reason why the non-restartable status is divided into red and brown is that the tool can explicitly inform programmers of the severity level of errors. Many of them are likely to cancel suspended refactorings under the brown status.

Actually, the Eclipse's refactoring tool performs two kinds of precondition checks before a programmer inputs information (called an initial check) and after the input action (called a final check). If she suspends EXTRACT METHOD after she already inputs some information (e.g., the name and the visibility of a newly extracted method) through the configuration dialog, the postponable refactoring tool stores the information. All of the refactoring settings necessary to restart are automatically restored from the stored information.

The significant benefit of use of the postponable refactoring tool is that restarting a suspended refactoring is lightweight, as compared with traditional refactoring tools that need troublesome reconfiguration when a programmer wants to apply a failed refactoring again. Moreover, awareness of the current status of a suspended refactoring could keep or raise her motivation for restarting it.

III. CONCLUSION

This paper proposed postponable automated refactoring and described a prototype tool [18] that supports the postponement of EXTRACT METHOD. We will improve the tool so that it can support other automated refactorings. In addition to improvement in usability, such a tool might be able to provide exact information about relationships between suspended, restarted, and canceled refactorings, which traditional refactoring tools can hardly manage. We will also make simulation-based evaluation and/or empirical study to demonstrate the performance and effects of postponable refactoring.

ACKNOWLEDGMENT

This work was sponsored by the Grant-in-Aid for Scientific Research (15H02685).

References

[1] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[2] E. Murphy-Hill and A. P. Black, "Refactoring tools: Fitness for purpose," *IEEE Software*, vol. 25, no. 5, pp. 38–44, 2008.

[3] E. Murphy-Hill, C. Parnin, and A. P. Black, "How we refactor, and how we know it," in *Proc. ICSE '09*, 2009, pp. 287–297.

[4] 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. ICSE '12*, 2012, pp. 233–243.

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

[6] M. Kim, T. Zimmermann, and N. Nagappan, "An empirical study of refactoring challenges and befits at Microsoft," *IEEE TSE*, vol. 40, no. 7, July 2014.

[7] S. R. Foster, W. G. Griswold, and S. Lerner, "WitchDoctor: IDE support for real-time auto-completion of refactorings," in *Proc. ICSE'12*, 2012, pp. 222–232.

[8] X. Ge, Q. L. DuBose, and E. Murphy-Hill, "Reconciling manual and automatic refactoring," in *Proc. ICSE'12*, 2012, pp. 211–221.

[9] X. Ge and E. Murphy-Hill, "Manual refactoring changes with automated refactoring validation," in *Proc. ICSE'14*, 2014, pp. 1095–1105.

[10] T. Mens and T. Tourwé, "A survey of software refactoring," *IEEE TSE*, vol. 30, no. 2, pp. 126–139, 2004.

[11] E. Murphy-Hill and A. P. Black, "Breaking the barriers to successful refactoring: Observations and tools for extract method," in *Proc. ICSE'08*, 2008, pp. 421–430.

[12] G. Soares, M. Mongiovi, and R. Gheyi, "Identifying overly strong conditions in refactoring implementations," in *Proc. ICSM'11*, 2011, pp. 173–182.

[13] M. Vakilian and R. E. Johnson, "Alternate refactoring paths reveal usability problems," in *Proc. ICSE'14*, 2014, pp. 1106–1116.

[14] ChangeMacroRecorder, `https://github.com/katsuhisamaruyama/ChangeMacroRecorder`.

[15] T. Omori and K. Maruyama, "A change-aware development environment by recording editing operations of source code," in *Proc. MSR'08*, 2008, pp. 31–34.

[16] Y. Yoon and B. A. Myers, "Supporting selective undo in a code editor," in *Proc. ICSE '15*, 2015, pp. 223–233.

[17] S. Negara, M. Vakilian, N. Chen, R. E. Johnson, and D. Dig, "Is it dangerous to use version control histories to study source code evolution?" in *Proc. ECOOP'12*, 2012, pp. 79–103.

[18] PostponableRefactoring, `https://github.com/katsuhisamaruyama/PostponableRefactoring`.