Scheduling of Conflicting Refactorings to Promote Quality Improvement

Hui Liu[1,2], Ge Li[1,2,*], Zhiyi Ma[1,2,*] , Weizhong Shao[1,2,*]
[1]Software Institute, School of Electronics Engineering and Computer Science
Peking University, Beijing 100871, P.R.China
[2]Key Laboratory of High Confidence Software Technologies (Peking University)
Ministry of Education, Beijing 100871, China
{liuhui04, lige, mzy, wzshao}@sei.pku.edu.cn

ABSTRACT

Software refactoring is to restructure object-oriented software to improve its quality, especially extensibility, reusability and maintainability while preserving its external behaviors. For a special software system, there are usually quite a few refactorings available at the same time. But these refactorings may conflict with each other. In other words, carrying out a refactoring may disable other refactorings. Consequently, only a subset of the available refactorings can be applied together, and which refactorings will be applied depends on the schedule (application order) of the refactorings. Furthermore, carrying out different subsets of the refactorings usually leads to different improvement of software quality. As a result, in order to promote the improvement of software quality, refactorings should be scheduled rationally. However, how to schedule refactorings is rarely discussed. Usually, software engineers carry out refactorings immediately when they are found out. They do not wait until all applicable refactorings are found out and scheduled. In other words, the refactorings are not scheduled explicitly, and conflicts among them are not taken into consideration. Though more and more refactorings are formalized and automated by refactoring tools, refactoring tools apply refactorings usually in a nondeterministic fashion (in random). In this paper, we propose a scheduling approach to schedule conflicting refactorings to promote the improvement of software quality achieved by refactorings. Conflicts among refactorings are detected, and then a scheduling model is presented. And then a heuristic algorithm is proposed to solve the scheduling model. Results of experiments suggest that the proposed scheduling approach is effective in promoting the improvement of software quality.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Restructuring, reverse engineering, and reengineering*; D.2.2 [**Software Engineering**]: Design Tools and Techniques

General Terms

Design, Algorithms

Keywords

Software Refactoring, Conflict, Schedule, Quality

*Corresponding Authors.

ASE'07, November 5–9, 2007, Atlanta, Georgia, USA.

1. INTRODUCTION

Software refactoring is to restructure object-oriented software to improve its quality, especially extensibility, reusability and maintainability while preserving its external behaviors [6]. It is an effective means to improve software quality [2]. According to new requirements, software is modified again and again. As a result, the source code (and the models) is jumbled, and the quality (especially maintainability) of the software is reduced [6]. So, further adaption to the software becomes more and more difficult and expensive. As an attempt to solve this problem, software refactoring was proposed [7], and object-oriented software (source code) was refactored to facilitate further adaption and extensions. Nowadays, software refactoring is not confined to source code any more, it has been applied to almost all types of software artifacts, including source code, designs, and software requirements [6]. Software refactoring is accepted as an important activity in software development and maintenance. In eXtreme Programming (XP) and other agile software development processes, refactoring is one of the core activities [8]. With the popularity of refactoring, more and more CASEs and IDEs, such as Eclipse and Visual Studio, begin to provide support for software refactoring. Professional refactoring tools, such as Refactoring Browser [9], are also developed. Tool support is crucial for successfully applying refactorings [4].

Software refactorings may conflict with each other [5]. One refactoring may change or delete elements that are necessary for other refactorings, and thus disable these refactorings. Conflicts among refactorings may be asymmetrical or symmetrical [5]. If a refactoring can be applied before but not after another refactoring, there is an asymmetrical conflict. If two refactorings can not be applied together in any order, the conflict is symmetrical. Conflicts among refactorings can be detected with formal analysis, such as critical pair analysis [5].

Different schedules of the same set of available refactorings may lead to different improvement of software quality. Though there are usually quite a few refactorings available for a specifical software system, it is very possible that only a subset of the available refactorings can be applied together because the refactorings may conflict with each other. And which refactorings will be applied depends on the application order (schedule) of the refactorings. Furthermore, carrying out different subsets of the refactorings usually leads to different improvement of software quality because different refactorings usually have different impact on software quality. Consequently, in order to achieve the greatest improvement of software quality, refactorings should be scheduled rationally, i.e. managing an appropriate application order of the available refactorings.

However, how to schedule refactorings is rarely discussed [1]. Usually, software engineers will carry out applicable refactorings immediately when they are found out. They usually do not wait until all applicable refactorings are found out and scheduled. In other words, the refactorings are not scheduled explicitly, and conflicts among the available refactorings are not taken into consideration. Though more and more refactorings are formalized and automated by refactoring tools, refactoring tools apply refactorings usually in a nondeterministic fashion (in random) [3].

In this paper, we propose a scheduling approach to schedule conflicting refactorings. The goal of the scheduling approach is to improve the software quality as much as possible by arranging an appropriate application order of the available refactorings.

The rest of this paper is structured as follows. Section 2 presents a motivating example to illustrate the necessity to schedule conflicting refactorings. Section 3 proposes a scheduling approach. Section 4 presents experiments of the proposed scheduling approach. Section 5 makes a conclusion.

2. MOTIVATING EXAMPLE

We take two refactoring rules into consideration: *Delete Dead Field* and *Pull Up Field*. If a field of a class is not used by the class or any subclass of it, the field is a dead field which should be deleted (*Delete Dead Field*). If all subclasses of class A contain the same field f, the field should be pulled up into the superclass (*Pull Up Field*). Suppose that the two refactoring rules have been formalized and automated by a refactoring tool, and the a program which contains three classes A, B and C (as shown in the top left corner of Fig. 1) is being processed by the refactoring tool. Furthermore, the public field *Va* is accessed by methods of class C, but not accessed by class B

Though the public field *Va* is useful for class C, i.e. accessed by methods of class C, it is definitely useless for class B. As a result, the refactoring rule *Delete Dead Field* can be applied to the field *Va* of class B, and the result of the refactoring is presented in the bottom left corner of Fig. 1.

It is also possible to apply refactoring rule *Pull Up Field* to the original program. Since both subclasses (B and C) of A contain public field *Va*, the common field should be pulled up to the superclass A according to refactoring rule *Pull Up Field*. The result of the refactoring is presented in the top right corner of Fig. 1.

As discussed above, either of the refactoring rules can be applied to the program. However, they can not be applied to the program together (as shown in the bottom right corner of Fig. 1). When the dead field *Va* of class B is removed (*Delete Dead Field*), it is impossible to pull field *Va* from class B any more (*Pull Up Field*); When the common field *Va* is pulled up from subclasses B and C (*Pull Up Field*), it is impossible to remove field *Va* from class B (*Delete Dead Field*) or A (because the pulled up field is still accessed by methods of C), either.

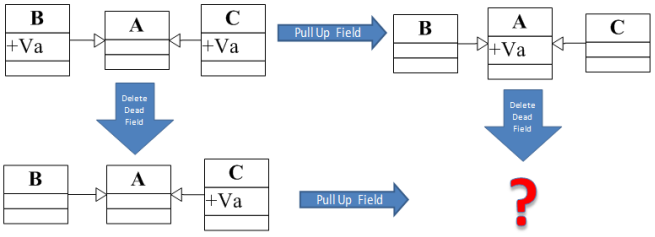

Figure 1: Conflicts between *Delete Dead Field* and *Pull Up Field*

In conclusion, the refactoring result depends on the application order of the two available refactorings. If *Pull Up Field* is applied first, the result is the model in the top right corner of Fig. 1. Otherwise, the result is the one in the bottom left corner.

3. SCHEDULING OF REFACTORINGS

3.1 Overview

In this section, we present a scheduling approach for conflicting refactorings. The goal of the scheduling approach is to promote quality improvement achieved by the application of the refactorings.

As the first step of scheduling, conflicts among refactorings should be detected. Section 3.2 proposes an approach to detect conflicts among refactorings. And then, Section 3.3 propose a scheduling model. The target of the scheduling model is the maximal quality improvement achieved by carrying out the selected refactorings. And the, a heuristic algorithm is proposed in Section 3.4 to solve the scheduling model.

3.2 Conflict Matrix

As the first step of the scheduling of conflicting refactorings, conflicts among refactorings should be detected.

Suppose there are two applicable refactorings, t_1 and t_2:

$$t_1 : \quad S \xrightarrow{r_1,m_1} S_1 \qquad t_2 : S \xrightarrow{r_2,m_2} S_2$$
$$r_1 : \quad L_1 \xrightarrow{f_1} R_1 \qquad r_2 : L_2 \xrightarrow{f_2} R_2$$

Refactoring t_1 will disable (asymmetric conflict) refactoring t_2 only if $m_1(L_1 - R_1) \cap m_2(L_2) \neq \emptyset$. The conflict may be symmetric or asymmetric. The algorithm to detect the conflict matrix of the refactorings is presented in Fig. 2. Every element $element_j$ involved in matches is scanned twice. In the first scan, the algorithm records the matches $element_j.NeededMatches$ which need (contain) the element, and the matches $element_j.DestroyedMatches$ by restructuring which the element will be destroyed. In the second scan, the algorithm detects conflicts according to the records made in the first scan: restructuring any match in $element_j.DestroyedMatches$ will destroy all the matches in

```
Input: ST = {t1, t2, t3 ··· tn}
For each ti ⊆ ST
   For each elementj ⊆ mi(Li)
      elementj.NeededMatches.add(i)
      If (elementj ∈ mi(Li − Ri))
         elementj.DestroyedMatches.add(i)
      End If
   End For
End For
For each ti ⊆ ST
   For each elementj ⊆ mi(Li)
      If (elementj.DestroyedMatches ≠ ∅)
         For each k ⊆ elementj.DestroyedMatches
            For each ℓ ⊆ elementj.NeededMatches
               C(k, ℓ) = 1
            End For
         End For
      End If
   End For
End For
For i ∈ [1, n]
   C(i,i)=0
End For
```

Figure 2: Detection of Conflict Matrixes

$element_j.NeededMatches$. At the end, $C(i,i)$ is set to zero because a refactorings will not conflict with itself.

3.3 Scheduling Model

All available refactorings and the conflicts among them are formalized as a directed graph G. Each vertex v_i represents a refactoring t_i. A directed edge from vertex v_i to v_j indicates that carrying out refactoring t_i will disable refactoring t_j. If there is also a directed edge from vertex v_j to v_i at the same time, there is a symmetric conflict between refactorings t_i and t_j.

For each refactoring t_i, experts will estimate the quality improvement achieved by carrying out the factoring t_i. The quality improvement of t_i is represented as Q_i.

The scheduling approach is to select a sequence of the available refactorings so as to maximize the quality improvement.:

$$\begin{cases} \max \quad QualityImprovement = \sum_{i\in[1,k]} Q_{x_i} \\ Seq \quad SQ = \langle t_{x_1}, t_{x_2}, t_{x_3}, \cdots t_{x_k} \rangle \end{cases} \tag{1}$$

s.t.

$$\begin{cases} 1 \le k \le N \\ 1 \le x_i \le N \quad i \subseteq [1,k] \\ x_i \ne x_j \qquad \text{if } i \ne j \\ \langle v_{x_i}, v_{x_j} \rangle \notin E \quad \text{if } i \le j \quad and \quad i,j \subseteq [1,k] \end{cases} \tag{2}$$

Where E is the edges of the directed graph G, and N is the size of G. SQ is a sequence of refactorings selected to be applied. $QualityImprovement$ is the total quality improvement achieved by carrying out the selected refactoring sequence. The last constrain

$$\langle v_{x_i}, v_{x_j} \rangle \notin E \quad \text{if } i \le j \quad and \quad i,j \subseteq [1,k]$$

guarantees that no selected refactoring will be disabled by the previously applied refactorings.

3.4 Heuristic Algorithm

Given a set of refactorings and their conflict matrix C:

$$\begin{cases} ST = \{t_1, t_2, t_3 \cdots t_N\} \\ C_{i,j} = \begin{cases} 1 & \text{if applying } t_i \text{ will disable } t_j \\ 0 & \text{else} \end{cases} \\ Size(C) = (N, N) \end{cases} \tag{3}$$

The following algorithm is given to schedule the refactorings:

Step 1: (Initializing G) Draw the initial directed graph $G = \langle V, E \rangle$ to represent the refactorings and conflicts among them. Where V and E are the vertexes and edges of G respectively. Every refactoring t_i is represented as a vertex v_i, and $e_{i,j} \in E$ if and only if $C_{i,j} = 1$

Step 2: (Initializing SQ) Initialize the restructuring sequence SQ to empty: $SQ = \langle\rangle$

Step 3: (Carrying out Uninjurious Refactorings) Vertexes without outgoing edges represent uninjurious refactorings which will not disable other refactorings. These refactorings are carried out first.

If $|G| = 0$, the algorithm ends.

If there is no uninjurious refactorings, go to Step 4.

Otherwise, if vertex v_i represents an uninjurious refactoring, v_i and its edges are removed, and the corresponding refactoring t_i is carried out ($SQ = SQ + t_i$). The quality improvement Q_i of the corresponding refactoring t_i is added to the $QualityImprovement$:

$$QualityImprovement = QualityImprovement + Q_i$$

where $QualityImprovement$ is the current value of the quality improvement. Go back to Step 3 to carry out the next uninjurious refactoring.

Step 4: (Computing Potential Quality Improvement) Carrying out a refactoring will not only directly impact the quality improvement. It also possible to disable other refactorings. This should be counted on as potential negative impact. Furthermore, carrying out refactoring t_i also reduces the potential negative impact of refactorings which have asymmetrical conflicts to t_i. The potential quality improvement PQ_i of refactoring t_i is computed in the following way:

$$PQ_i = Q_i - \sum_{e_{i,j}\in E} Q_j + \sum_{e_{\ell,i}\in E} Q_\ell \tag{4}$$

Step 5: (Selection and Application) Select a vertex v_i from G which has the greatest potential quality improvement PQ_i. Carry out the corresponding refactoring t_i, update the quality improvement.

$$QualityImprovement = QualityImprovement + Q_i$$

If no vertex is left, the algorithm ends.

Step 6: (Update Potential Quality Improvement) Once the selected refactoring is carried out, the corresponding vertex v_i should be removed from the G. Before that, we should update the potential quality improvement for vertexes adjacent to v_i.

For each $e_{i,j} \in E$, i.e. there is an edge from v_i to v_j, remove v_j and its edges from the directed graph. The edge from v_i to v_j indicates that the application of refactoring t_i will disable refactoring t_j. As a result, once t_i is carried out, refactoring t_j should be removed from candidate (available) refactorings.

For each $e_{j,i} \in E$, i.e. there is an edge from v_j to v_i:

$$PQ_j = PQ_j + Q_i \tag{5}$$

Once v_i is removed, the potential quality improvement of v_j should be updated according to Formula 4. The result of the update is equivalent to Equation 5.

Go back to Step 5.

4. EXPERIMENTS

We apply the proposed scheduling approach to refactorings on an ongoing research project: PKU Meta-Modeling Tool (PMMT). As indicated by the name, PMMT is a meta-modeling tool developed in Peking University.

The purpose of the experiments is to compare the new refactoring approach, in which refactoring conflicts are explicitly considered and conflicting refactorings are scheduled with the proposed scheduling algorithm, with the traditional refactoring approach, in which scheduling is not explicitly proposed. So, we decide that both of the refactoring approaches should be applied to the evaluation project (two copies of the project in fact) independently. Once refactoring activities are completed, the two refactored projects are compared by experts.

In the first experiment, the traditional refactoring approach is applied. Experts looked around, found bad smells, restructured the bad smells, and then went ahead for more bad smells.

In the second experiment, experts found out and estimate all bad smells (but no refactoring was carried out in this stage). And then the proposed scheduling approach was applied to these candidate (available) refactorings to detect the conflicts among them, and then managed a schedule (application order). After that, selected refactorings were carried out according to the schedule.

The resulting systems were evaluated by five experts, and the results of the evaluation are presented in Fig. 3. Each expert would give a number ranging from 0 to 10 to indicate the quality of the resulting system, where 0 indicates the worst and 10 is perfect.

The results suggest that scheduling of conflicting refactorings would result in greater quality improvement. As suggested by Fig. 3, all experts agreed that the resulting system of the second experiment (refactoring with scheduling) is better than that of the first experiment (without explicit scheduling).

5. CONCLUSION

Since refactorings may conflict, which refactorings will be applied depends on the schedule of the refactorings. And different refactorings usually have different impact on software quality. In this paper, we propose a scheduling approach to schedule conflicting refactorings to promote quality improvement. The results of the experiments suggest that the proposed scheduling approach is effective in promoting quality improvement.

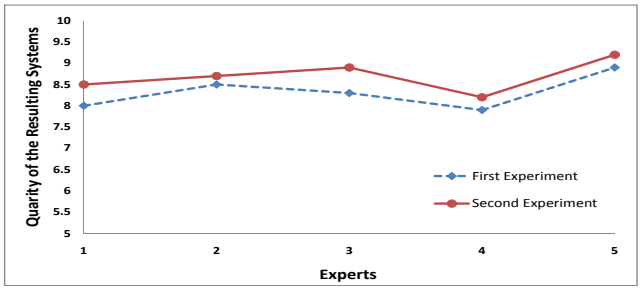

Figure 3: Comparison of Resulting Systems

6. ACKNOWLEDGEMENTS

The work is funded by the National Grand Fundamental Research 973 Program of China No. 2005CB321805, the National Natural Science Foundation of China No. 60473064,the National High-Tech Research and Development Plan of China No. 2007AA01Z127, and National Key Technology R&D Program of China No. 2006BAH02A02

7. REFERENCES

[1] S. Bouktif, G. Antoniol, E. Merlo, and M. Neteler. A novel approach to optimize clone refactoring activity. In *Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation*, pages 1885–1892, 2006.

[2] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison Wesley Professional, 1999.

[3] T. Mens. On the use of graph transformations for model refactoring. Slides of a tutorial at the Summer School on Generative and Transformational Techniques in Software Engineering, 2005.

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

[5] T. Mens, G. Taentzer, and O. Runge. Detecting structural refactoring conflicts using critical pair analysis. *Electronic Notes in Theoretical Computer Science*, 127(3):113–128, April 2005.

[6] T. Mens and T. Touwe. A survey of software refactoring. *IEEE Transaction on Software Engineering*, 30(2):126–139, 2004.

[7] W. F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[8] J. U. Pipka. Refactoring in a 'test first'-world. In M. Marchesi and G. Succi, editors, *Third International Conference on eXtreme Programming and Flexible Processes in Software Engineering*, pages 178–181, 2002.

[9] D. Roberts, J. Brant, and R. Johnson. A refactoring tool for smalltalk. *Theory and Practice of Object Systems, Special issue object-oriented software evolution and re-engineering*, 3(4):253 – 263, 1997.