Front. Comput. Sci., 2016, 10(4): 644–656
DOI 10.1007/s11704-016-5131-4

RESEARCH ARTICLE

Major motivations for extract method refactorings: analysis based on interviews and change histories

Wenmei LIU, Hui LIU (✉)

School of Computer Science and Technology, Beijing Institute of Technology, Beijing 100081, China

Abstract Extract method is one of the most popular software refactorings. However, little work has been done to investigate or validate the major motivations for such refactorings. Digging into this issue might help researchers to improve tool support for extract method refactorings, e.g., proposing better tools to recommend refactoring opportunities, and to select fragments to be extracted. To this end, we conducted an interview with 25 developers, and our results suggest that current reuse, decomposition of long methods, clone resolution, and future reuse are the major motivations for extract method refactorings. We also validated the results by analyzing the refactoring history of seven open-source applications. Analysis results suggest that current reuse was the primary motivation for 56% of extract method refactorings, decomposition of methods was the primary motivation for 28% of extract method refactorings, and clone resolution was the primary motivation for 16% of extract method refactorings. These findings might suggest that recommending extract method opportunities by analyzing only the inner structure (e.g., complexity and length) of methods alone would miss many extract method opportunities. These findings also suggest that extract method refactorings are often driven by current and immediate reuse. Consequently, how to recognize or predict reuse requirements timely during software evolution may play a key role in the recommendation and automation of extract method refactorings. We also investigated the likelihood for the extracted methods to be reused in future, and our results suggest that such methods have a small chance (12%) to be reused in future unless the extracted fragment could be reused immediately in software evolution and extracting such a fragment can resolve existing clones at the same time.

Keywords software refactoring, extract method, motivation, data mining, software quality

1 Introduction

Software refactoring is to restructure software applications without changing their external behaviors [1, 2]. Software refactoring is popular and 41% of programming sessions contain refactoring activities [3]. Analysis results on Eclipse code base [4] also suggest that software refactoring is popular and about 70% of structural changes are caused by software refactorings.

Extract method is one of the most popular software refactorings. Extract method is to extract some statements, e.g., a fragment of source code or a slice of source code from an existing method as a new method [5–8]. For convenience, we call the new method *extracted method* in this paper. Investigations conducted by Palomba et al. [9] suggest that the code smell usually associated with extract method refactoring, i.e., long methods, is almost perceived by developers, and it is considered as the most harmful from their point of view. The finding may suggest that researchers should work more on extract method refactorings. According to [10], among dozens of software refactorings *extract method* is the fourth popular. Most of the mainstream IDEs, e.g., *Eclipse*, *IntelliJ IDEA*, and *Visual Studio*, provide automated tool support for extract

Received April 2, 2015; accepted November 10, 2015

E-mail: Liuhui08@bit.edu.cn

method refactorings. Such refactoring tools can automatically extract a fragment of source code that is selected manually by developers whereas the functionality of the involved application is not changed.

Although extract method refactoring is popular, little work has been done to investigate or validate the major motivations for such refactorings. As a result, little is known about the predominant motivations for such refactorings and how many extract method refactorings are conducted for these motivations. Digging into these issues might help researchers better understand extract method refactorings. Besides that, digging into these issues might also help to improve tool support for extract method refactorings, e.g., proposing better tools to recommend refactoring opportunities, and to select fragments to be extracted.

To this end, we interviewed 25 developers, and the results suggest that decomposition of long methods, current reuse, clone resolution, and future reuse are the major motivations for extract method refactorings. We validated the results by analyzing the refactoring history of seven open-source applications. Analysis results suggest that current reuse was the primary motivation for 56% of extract method refactorings, decomposition of methods was the primary motivation for 28% of extract method refactorings, and clone resolution was the primary motivation for 16% of extract method refactorings. We also investigated the likelihood of extracted methods to be reused in future, i.e., in versions later than the one where the extracted methods were introduced. The results suggest that only 17% of such methods have been reused in future versions. However, if the extracted fragment could be reused immediately in software evolution and extracting such a fragment could resolve existing clones at the same time, the resulting new methods would have much higher likelihood (39%) to be reused in future than new methods generated by other extract method refactorings (12%).

Researchers and developers might be interested in these findings because such findings suggest that:

- Recommending extract method opportunities by analyzing only the inner structural(e.g., complexity, length, and cohesion) of methods might miss many extract method opportunities. Our findings suggest that more than 70% of extract method refactorings are conducted for other reasons, e.g., to remove clones and to facilitate current and immediate reuse.
- Extract method refactorings are often driven by current reuse requirements. Our results suggest that current and immediate reuse is the primary motivation for 56% of extract method refactorings. Consequently, how to recognize or predict reuse requirements timely during software evolution may play a key role in the recommendation and automation of extract method refactorings. However, to the best of our knowledge, existing refactoring tools cannot recognize or predict reuse requirements timely. As a result, existing tools fail to identify such refactoring opportunities that should be conducted to facilitate current reuse requirements. Failing to recommend such refactoring opportunities may lead to fewer or delayed (if identified later) refactorings [11] because inexperienced developers may not be able to find such refactoring opportunities manually and timely. It might also suggest that requirements driven software refactoring [12] could be a potential research direction.
- Extracted methods have a small chance to be reused in future. Our results suggest that on average only 17% of such methods are reused in future. Consequently, extracting fragments for future reuse without concrete current reuse requirements might be unproductive. However, our results also suggest that if the extracted fragments are reused immediately in software evolution and extracting such a fragments can resolve existing clones at the same time, the resulting extracted methods have a much higher chance (39%) to be reused in future. In contrast, only 12% of new methods that were generated by other extract method refactorings have been reused in future. Practitioners of extract method refactorings should think twice on the costs and benefits of such refactorings, if future reuse is one of the motivations (and expected benefits) for such refactorings.

In summary, this paper contributes the following:

- An interview on the motivations for extract method refactoring.
- An extensive empirical study of motivations for extract method refactoring.

Developers and researchers in this field may be interested in these contributions as discussed in the preceding paragraph, which makes the contributions substantial.

The rest of the paper is structured as follows. Section 2 reviews related research. Section 3 describes and contextualizes our qualitative analysis of extract methods' motivations via interviews. Section 4 uses seven open-source applications'

change histories to validate and to offer quantitative insights into the findings. Section 5 discusses threats to validity. Section 6 provides conclusions and potential future work.

2 Related work

To put software refactoring research on sound scientific basis, a number of researchers have investigated how software refactorings are conducted. Tsantalis et al. [13] manually analyzed 210 extract method refactorings to investigate the motivations for such refactorings. The analysis suggests that extract method refactorings are driven by a number of different motivations. The results also suggest the decomposition of methods is the predominant motivation. Our research differs from theirs in the following aspects. Firstly, we have different definition of extract method refactorings. For example, *encapsulate filed* was taken as extract method refactorings in [13] whereas it is excluded in this paper because the target of *encapsulate filed* is a field and it should change all associated field access in different methods. Fowler [14] took *encapsulate filed* and *extract method* as two different refactorings as well. Secondly, their analysis was conducted manually whereas ours was automated. It is challenging to analyze thousands of extract method refactorings manually although manual analysis could be more accurate. Thirdly, we have conducted interviews to investigate major motivations that were validated on refactoring history. Finally, new and interesting findings are presented in this paper, e.g., up to 67 percentages of extract method refactorings are conducted for current reuse, and the inability of recommenders purely based on structural properties to find all the extract method refactoring opportunities. We also find that new methods resulting from extract method refactorings have small chances to be reused in future unless such methods are reused immediately.

Murphy-Hill et al. [3] analyzed four data sets to investigate how refactorings were conducted. The first data set was collected by Mylyn Monitor tool from 41 volunteer developers [15]. The second one was 240 000 refactorings collected by Eclipse Usage Collector[1)]. The third one was refactoring histories from four maintainers of Eclipse refactoring tools, and the last one was 3 400 commits in Eclipse CVS. With the analysis, Murphy-Hill et al. [3] confirmed some assumptions, e.g., software refactoring is common. They also cast doubt on a number of previously stated assumptions, e.g., refactoring activities can be discovered by commit logs. Kim et al. [16] presented a field study of refactoring benefits and challenges. Their survey conducted in Microsoft suggests that refactoring definition in practice is not confined to rigorous semantics-preserving code transformations. Following these works, in this paper we investigated a narrowed but deeper refactoring issue, i.e., why developers conduct extract method refactorings.

Bavota et al. [17] conducted an experimental investigation to reveal the relationship between quality metrics of source and refactorings conducted on such code. Their findings suggest that more often than not, quality metrics (or code smells that are based on such metrics) do not show a clear relationship with refactorings. It is interesting that a number of existing approaches to identification of refactoring opportunities are based on metrics. Such approach, according to the finding, may be not effective. Palomba et al. [18] tried to identify refactoring opportunities by mining version histories, and results show that traditional structural information based approach miss some refactoring opportunities. Palomba [19] identified long methods with textual analysis, which is a successful try to use non-structural information to identify refactoring opportunities. Non-structural information is also employed by other approaches to identification of refactoring opportunities [20–22]. The findings introduced in this paragraph are in line with ours presented in this paper. Our work differs from such approaches in that we analyze the motivations for extract method refactorings whereas they identify refactoring opportunities.

Recovery of refactorings serves as a basis of refactoring analysis. Demeyer et al. [23] first proposed the idea of discovering refactorings by comparing two program versions. They have proposed and evaluated four heuristics for detecting refactorings that occurred between two versions. Xing et al. [24, 25] compared class diagrams generated by reverse engineering, and identified design level refactorings. Weissgerber and Diehl [26] analyzed transactions, i.e., all revisions checked in by the same commit operation. Among such transactions, they identified potential refactoring operations and ranked such operations with clone detection. Dig et al. [27] matched elements in different versions using shingles [28], and inferred refactorings by semantic analysis. Their approach has been implemented and the tool Refactoring Crawler is publicly available[2)]. Prete et al. [29] proposed a template-based approach to identify refactorings occurred between two program versions. The approach has been implemented and the tool REF-FINDER is publicly available as

[1)] https://www.eclipse.org/epp/usagedata/
[2)] http://dig.cs.illinois.edu/tools/RefactoringCrawler/

well[3]. Such tools might be used to discover extract method refactorings as well, and their approaches are similar to that in Section 4.3. However, these tools cannot classify extract method refactorings or reveal the motivations for such refactorings.

Origin analysis proposed by Godfrey and Zou [30] is a basis of refactoring discovery. Origin analysis is to match code elements, e.g., methods, using multiple criteria including names, signatures of such elements. One of the key challenges in original analysis is that names of software elements might change. To overcome this challenge, Kim et al. [31] matched methods by computing the similarity between methods. Malpohl et al. [32] identified renamed elements by comparing their similarity in declaration, implementation and references. Fluri et al. [33] proposed Change Distiller that compares two versions of abstract syntax trees and extracts changes in hierarchically structured data. Kim et al. [34] represented structural changes as a set of high-level change rules. They proposed an approach to automatically infer potential change rules and to match methods based on such rules. In this paper, we discovered extract method refactorings based on method-level matches generated by their approach.

3 Interviews

3.1 Overview

Overview of the research method is presented in Fig. 1. We first qualitatively analyzed the motivations for extract method refactorings via interviews. Such motivations were then validated on open-source applications' change histories.

To validate the motivations, we first discovered extract method refactorings by analyzing the evolutionary history of subject applications. With existing technologies and tools, we discovered these refactorings by comparing every pair of two consecutive versions of the same application. We then distinguished the application scenarios for these discovered refactorings, and classified these refactorings according to their application scenarios. With statistical data of the classified refactorings, we made some interesting observations and analyzed the potential implications of such observations.

3.2 Interviews design

We recruited 25 developers in total to understand their motivations of performing extract method refactorings. We interviewed each of them one by one by asking the following questions (in Chinese):

- Have you ever conducted extract method refactorings?
- What were the major motivations for you to conduct these extract method refactorings?

If a participant suggested that she/he has not ever conducted extract method refactorings or she/he did not understand the concept of extract method refactoring, the interview ended and she/he was not counted in.

If reuse was mentioned as one of the major motivations, the participant was asked the following additional question:

- Do you mean current and immediate reuse or potential future reuse?

If multiple motivations were mentioned by the same participant, she/he was asked to rank such motivations with the predominant one on the top.

Our 25 interviewees included 14 developers from the industry and 11 students who had worked in the industry before they came back to campus for master degrees. Industry participants were recruited by an electronic recruitment

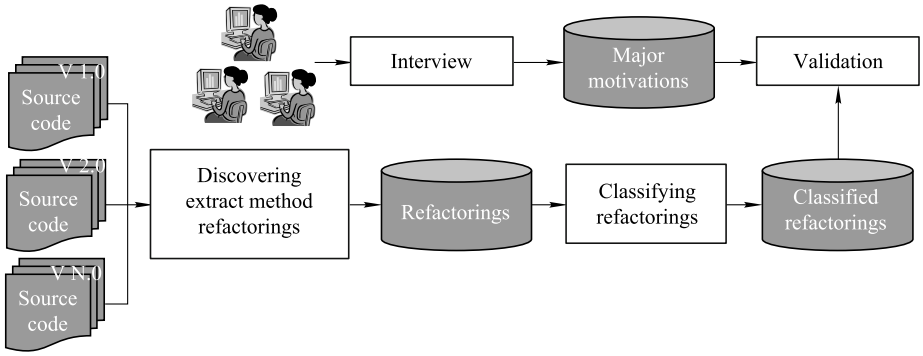

Fig. 1 Overview of the research method

[3] http://users.ece.utexas.edu/~miryung/software.html

email forwarded by our industry contacts within two companies. Eight of them were working in a mobile company, developing mobile applications. Six of them were working in a small company in the field of Internet of Things (IOT). Students with industry experience were recruited by posts on the campus. Participants' years of development experience ranged from three to eight years. The interview with each participant lasted around five to ten minutes.

3.3 Interview results

Twenty-four out of the twenty-five participants had conducted extract method refactorings. A participant was excluded because he declared that he had not yet conducted extract method refactorings. By analyzing the interview records, we classified the mentioned motivations as *current reuse*, *decomposition of methods*, *clone resolution*, *future reuse*, and *others*. *Others* included all other motivations except the four major ones mentioned above.

- Decomposition of methods
  Extract method might be conducted to decompose long and complex methods [14]. Such methods are usually difficult to read or modify, and thus decomposing such methods with extract method refactorings could help to improve software readability and maintainability.
- Clone resolution
  Extract method might be conducted to remove duplicate code. Negative impact of duplicate code and clones has been investigated by [35], and a number of tools have been proposed to identify duplicate code and clones [36–40]. Extracting duplicate code as a modularized method is an effective means to remove duplicate code.
- Current reuse
  Extract method might be conducted to facilitate current reuse during software evolution. While programming, developers might realize that an existing fragment of source code could be reused. Extracting this fragment as a reusable method enables developers to reuse the code by a single method call instead of by implementing the same function again. This scenario is different from the clone resolution with extract method refactorings in that in this scenario refactorings are driven by current and immediate reuse requirements whereas in the latter one such refactorings are driven by clone removals.
- Future reuse
  In some cases, a code fragment might be extracted because the fragment is likely to be reused in future. In other cases, however, facilitating future reuse is a secondary motivation. For example, decomposition of long methods with extract method is often driven by the complexity of such long methods (i.e., the major motivation is decomposition of methods) whereas facilitating future reuse is a secondary motivation for (and a secondary benefit of) the refactorings.

Results of the interview are presented in Table 1. From the table, we observe that:

- Current reuse was the predominant motivation for most participants. All of the 24 participants listed it as one of the major motivations, and 14 of them put it on the first place of their ranked list.
- All of the 24 participants listed decomposition of methods as one of the major motivations, and seven participants listed it as the predominant motivation.
- Sixteen out of the 24 participants listed clone resolution as one of the major motivations, and two participants listed it as the predominant motivation.
- Fifteen out of the 24 participants listed future reuse as one of the major motivations, and one participant listed it as the predominant motivation.
- Four out of the 24 participants listed others as one of the major motivations, but it was never listed as the predominant motivation. Two participants stated that sometimes they extracted a code fragment as a method because the fragment was likely to change independently. Two participants stated that sometimes they extracted methods because the extracted source code should be relocated to other classes.

Table 1 Interview results

Motivation	Mentioned	Mentioned as the predominant motivation
Current reuse	24 (100%)	14 (58%)
Decomposition of methods	24 (100%)	7 (29%)
Clone resolution	16 (67%)	2 (8%)
Future reuse	15 (63%)	1 (4%)
Others	4 (17%)	0 (0%)

We conclude from these results that extract method refactoring is popular, and from the viewpoint of developers the major motivations for such refactorings are current reuse, decomposition of methods, clone resolution, and future reuse. We also conclude from these results that the priority of such

motivations varies from person to person.

4 Refactoring history based validation

4.1 Research questions

To validate the interview results, the validation in this section investigates the following research questions:

- RQ1: What percentage of extract method refactorings are conducted for the major motivations, respectively?
- RQ2: What are the chances that new methods resulting from extract method refactorings are reused in future?

The first research question would give a quantitative validation of the interview results. The investigation of this question is interesting for developers of extract method refactoring tools, because they usually employ different approaches to recommend extract method refactoring opportunities that are driven by different motivations. For example, for those driven by decomposition of methods, it is common to identify such refactoring opportunities by measuring the length and complexity of methods. In contrast, for those driven by current reuse, it is more practical to recommend refactoring opportunities by analyzing new requirements and their relationship with existing source code.

The second research question is interesting for practitioners of extract method refactorings. As suggested by the interview results, 63% of the practitioners take future reuse as one of the major benefits of extract method refactorings. Quantitative analysis of the second research question would reveal to what extent the benefit could be achieved in practice. If the analysis of refactoring history suggests that new methods generated by extract method refactorings are rarely reused in future, the practitioners might think twice on the costs and benefits of such refactorings, especially when future reuse is the single (or major) motivation for such refactorings.

4.2 Subjects

As declared in Section 3.1, we discovered extract method refactorings by analyzing the evolutionary history of open-source applications. We selected seven popular open-source applications from *SourceForge* to discover extract method refactorings on such applications. An overview of these applications is presented in Table 2, and details are presented as follows:

- *FindBugs*[4] is a bug detection tool. It uses static analysis to identify bugs in Java code. Source code of this application was downloaded from SourceForge[5]. We analyzed the last nine versions (from version 1.3.5 to version 3.0.0). Version 1.3.5 was released on 2008/9/13 whereas version 3.0.0 was released on 2014/7/7. The size of FindBugs varies from 81 563 to 123 259 LOC (varying from from version to version).
- *Hibernate*[6] is an open-source application developed in Red Hat[7]. The purpose of this project is to provide an easy way to achieve persistence in Java. Source code of this application was downloaded from SourceForge[8]. We analyzed the last 31 versions (from version 4.0.0 to version 5.6.1). Version 4.0.0 was released on 2004/12/20 whereas version 5.6.1 was released on 2014/4/8. The size of the application varies from 68 929 to 176 879 LOC.
- *JEdit*[9] is a mature programmer's text editor. It provides auto indent and syntax highlighting for more than 200 languages. Source code of this application was downloaded from SourceForge[10]. We analyzed the last 17 versions (from version 2.3 to version 5.1.0). Version 2.3 was released on 2000/2/1 whereas version 5.1.0 was released on 2013/7/29. The size of the application varies from 23 881 to 115 919 LOC.
- *JFreeChart*[11] is a pure Java chart library. It is the most widely used chart library for Java, and it has been downloaded more than 2.2 million times. Source code of this application was downloaded from SourceForge[12]. We analyzed the last 35 versions (from version 0.9.0 to version 1.0.18). Version 0.9.0 was released on 2002/6/7 whereas version 1.0.18 was released on 2014/7/3. The size of the application varies from 6 978 to 99 375 LOC.

[4] http://findbugs.sourceforge.net/
[5] http://sourceforge.net/projects/findbugs
[6] http://hibernate.org/
[7] http://www.redhat.com/en
[8] http://sourceforge.net/projects/hibernate
[9] http://www.jedit.org/
[10] http://sourceforge.net/projects/jedit
[11] http://www.jfree.org/JFreeChart/
[12] http://sourceforge.net/projects/JFreeChart/

- *Weka*[13] is an open-source application developed by the machine learning group at the University of Waikato[14]. It implements in Java a set of well-known machine learning algorithms. Source code of this application could be downloaded from its SVN server[15]. We analyzed the last 38 versions (from version 3.1.7 to version 3.7.11). Version 3.1.7 was released on 2000/5/1 whereas version 3.7.11 was released on 2014/4/29. The size of the application varies from 38 890 to 272 212 LOC.
- *JasperReports*[16] is an open-source Java reporting tool. It is composed of two parts, JasperReports Library and JasperReports Server. It is one of the most popular open-source business intelligence and reporting engines. We analyzed the last 19 versions (from version 4.0.0 to version 5.6.1). Version 4.0.0 was released on 2011/1/10 whereas version 5.6.1 was released on 2014/9/4. The size of the application varies from 195 135 to 268 520 LOC.
- *Vuze-Azureus*[17] is a peer to peer (P2P) file sharing client using the bittorrent protocol. It also converts videos and music for playing on non-PC devices, e.g., PSP, XBox, and iPhone. We analyzed the last 21 versions (from version 4402 to version 5400). Version 4402 was released on 2010/5/5 whereas version 5400 was released on 2014/8/3. The size of the application varies from 482 134 to 574 638 LOC.

Table 2 Subject applications

Application	Domain	Number of versions	Size (LOC)
FindBugs	Bug detection	9	81 563–123 259
Hibernate	Java persistence	31	68 929–176 879
JEdit	Text editor	17	23 881–115 919
JFreeChart	Chart library	35	6 978–99 375
Weka	Machine learning	38	38 890–272 212
JasperReports	Java reporting tool	19	195 135–268 520
Vuze-Azureus	File sharing	21	482 134–574 638

We selected such subjects because of the following reasons. First of all, they are open-source and thus we can analyze their source code. Second, all of such applications have a long evolutionary history which enables the discovery of extract method refactorings on these applications. Third, these applications are well-known and popular. Finally, these applications are from different domains and were developed by different people. Overall, we expect our subject selections to help mitigate certain threats to external validity and reproducibility.

4.3 Discovery of extract method refactorings

For two successive versions (v_1 and v_2, respectively) of the same application, the discovery of extract method refactorings was composed of two stages. In the first stage, we identified methods added to the later version (v_2). Among such new methods, in the second stage we identified those that had been added as a result of extract method refactorings.

To identify new methods, we matched methods in two successive versions according to the matching algorithm proposed by Kim et al. [34]. The matching algorithm was selected because of the following reasons. Firstly, the algorithm has been implemented and its source code is publicly available[18]. Secondly, it is effective and efficient. With this algorithm, we obtained a list of matched methods, notated as M_{match}, and a list of methods (notated as M_{new}) that appear in v_2 but not in v_1. Each element in M_{match} consists of two methods m_{old} and m_{new} that appear in v_1 and v_2, respectively.

For each method m in M_{new}, it represents an extracted method refactorings if

- it is invoked by at least one method;
- at least one of its callers (c_1) has a matching method c_2 in the earlier version, i.e., $\langle c_1, c_2\rangle \in M_{match}$;
- the body of the new method m is similar to the fragment of c_2 that does not appear in c_1. We note this fragment as $fg = c_2 - c_1$.

The similarity between the new method m and the fragment fg is computed as follows:

$$Sim(m, fg) = \frac{|\text{common}(m, fg)|}{|m|}, \tag{1}$$

where $|m|$ is the number of lines within method m, and $|\text{common}(m, fg)|$ is the number of common lines between m and fg. We identify the common lines of source code between m and fg with Diff[19]. We use Diff because it is effi-

[13] http://sourceforge.net/projects/weka/
[14] http://www.cs.waikato.ac.nz/ml/weka/
[15] https://svn.cms.waikato.ac.nz/svn/weka
[16] http://sourceforge.net/projects/JasperReports/
[17] http://sourceforge.net/projects/azureus/
[18] http://www.cs.ucla.edu/~miryung/software.html
[19] http://code.google.com/p/java-diff-utils/

cient and its Java implementation is publicly available. Since our implementation is coded in Java, it is easy to reuse the Java implementation of Diff. However, as discussed by Canfora et al. [41] and Asaduzzaman et al. [42], the Diff algorithm contains some limitations in detecting reordered or changed lines. In future, we would like to try and compare more advanced line differencing algorithms, like *Ldiff* [41] and *LHDiff* [42].

If the similarity $Sim(m, fg)$ is greater than a threshold β, they are declared as similar. Otherwise, they are not similar. We calibrate the approach on a real-world application that contains 20 extract method refactorings, and conclude that $\beta = 0.5$ can achieve a good balance between recall and precision [43]. The calibration depends on recall that in turn depends on the accurate number of true positives (conducted extract method refactorings). Consequently, the calibration cannot be conducted on large scale open-source applications, because we cannot identify all of the extract method refactorings conducted on such applications. The subject application used in the calibration is middle-sized, and all extract method refactorings have been recorded and thus facilitate the calibration. The resulting threshold $\beta = 0.5$ is relatively small because of the following reasons. First, when a fragment is extracted as a new method, it is likely for developers to make changes on the extracted source code, e.g., extracting parameters and generalizing the extracted source code to improve its reusability. As a result, the similarity may be reduced, especially when the length of the extracted source code is small. Second, similarity between m and fg is only one of the preconditions in deciding whether m represents an extract method refactoring. Consequently, a relatively small threshold in the similarity may also result in high precision.

The corresponding extract method refactoring is represented as $r = \langle m, c_1, c_2 \rangle$, suggesting that the refactoring is to extract a fragment from method c_2 as a new method m.

4.4 Classification

For extract method refactorings $R = \{r_1, r_2, \ldots, r_n\}$ discovered by the previously described method, we classified them into the following categories:

- $Type_1$ $r = \langle m, c_1, c_2 \rangle$ belongs to this category if c_1 is the only method that invokes the new method m. Extract method refactorings that are conducted to decompose methods should result in a new method that is called by a single method (where the new method is extracted). In contrast, extracting method for current reuse or for clone resolution should result in multi-invocation of the new method.
- $Type_2$ $r = \langle m, c_1, c_2 \rangle$ belongs to this category if 1) more than one method invokes the new method m; 2) all such callers have matching methods in the earlier version; and 3) all such matching methods in the earlier version contain a fragment that is similar to the body of the new method m. In other words, all such matching methods in the earlier version share a similar fragment extracted as the new method. Consequently, refactorings of $Type_2$ were conducted to resolve clones, and the new method has not been reused in implementation of new requirements.
- $Type_{3A}$ $r = \langle m, c_1, c_2 \rangle$ belongs to this category if 1) more than one method invokes the new method m; and 2) only one of such callers has a matching method in the earlier version that contains a fragment similar to the body of the new method m. According to the second condition, we know that refactoring r is not conducted to resolve clones. It is likely that the refactoring is conducted to facilitate current reuse because m is called by methods that do not contain the extracted source code in the earlier version. It is also theoretically possible that the refactoring is conducted to decompose methods ($Type_1$ refactoring) and then the new method is reused immediately on the same version. However, according to the study in Section 4.3.3, the likelihood of a new method (resulting from a $Type_1$ refactoring) to be reused at a specific version is less than 2%. Consequently, we roughly classified refactorings of $Type_{3A}$ as extract method refactorings whose primary motivation was for current reuse.
- $Type_{3B}$ $r = \langle m, c_1, c_2 \rangle$ belongs to this category if 1) n_1 ($n_1 > 2$) methods invoke the new method m; and 2) n_2 ($1 < n_2 < n_1$) of such callers has a matching method in the earlier version that contains a fragment similar to the body of the new method m. According to the second condition, we know that one of the reasons why refactoring r is conducted is to resolve clones. However, according to the first condition, we also know that another reason why it is conducted is to facilitate current reuse. However, according to the study in Section 4.3.3, the likelihood of a new method (resulting from a $Type_2$ extract method refactoring) to be reused at a specific version is less than 4%. Consequently, current reuse is the primary motivation for $Type_{3B}$ extract method refactorings whereas clone resolution might be another potential secondary motivation.

According to the analysis in the preceding paragraphs, the four categories of extract method refactorings represent three refactoring scenarios. Extracting method refactorings motivated by decomposition of methods belong to $Type_1$, those motivated by clone resolution belong to $Type_2$, and those whose predominant motivation was for current reuse belong to $Type_3$ (including $Type_{3A}$ and $Type_{3B}$).

The approach to discover and classify extract method refactoring has been implemented as an Eclipse plug-in, named as *Extract Method Detector*. The source code is available at http://www.sei.pku.edu.cn/~liuhui04/tools/exDetector.htm. According to our initial evaluation, its precision in detecting extract method refactorings is around 90% whereas its recall is around 85% [43]. For space limitation, the technical details and initial evaluation of the approach is presented in a technical report [43].

4.4.1 Results

Extract Method Detector discovered 1 619 extract method refactorings from the selected open-source applications. Details are presented in Table 3. As suggested by the table, 117, 160, 271, 303, 238, 346, and 184 extract method refactorings were discovered from FindBugs, Hibernate, JEdit, JFreeChart, Weka, JasperReports, and Vuze-Azureus, respectively.

Table 3 Discovered extract method refactorings

Application	Number of extract method refactorings
FindBugs	117
Hibernate	160
JEdit	271
JFreeChart	303
Weka	238
JasperReports	346
Vuze-Azureus	184
Total	1 619

Among the 1 619 extract method refactorings, three postgraduate students majored in computer science selected 140 refactorings in random. They manually checked the selected refactorings, and confirmed that 127 of them were real extract method refactorings whereas the other 13 were false positives. The precision of Extract Method Detector on the samples was $91\% = 127/140$. It is difficult, if not impossible, to achieve a precision of one hundred percents. The reason is explained as follows. It is quite often that the new method resulting from an extract method refactoring is slightly different from the source code extracted during the refactoring, because some modifications are required to adapt the code to its new context. Consequently, to improve the recall of the approach in identifying extract method refactorings, the approach identifies extract method refactorings by computing the similarity between new methods in the latter versions and missing fragments in the elder versions (see Section 4.3 for details). If the similarity is greater than a predefined threshold (instead of requiring them to be identical), the approach would suggest that the new methods are introduced by extract method refactorings. However, it is possible that a fragment of source code is removed for some reasons other than extracting methods, and a new method containing similar source code is added. In this case, a false positive (inaccuracy) would be reported. Murphy-Hill et al. [44] has discussed some alternative approaches to collecting refactorings, and conclude that each approach has its strength and weakness. Collecting refactorings by observing developers may reduce the inaccuracy. However, large scale observation is time and resource consuming [44].

We conclude from these results that extract refactoring is popular, and it is worthy to investigate the motivations for such refactorings if the investigation can improve the development of powerful and efficient tools for extract refactoring.

4.4.2 Results of classification (answering RQ1)

The results of the classification of discovered extract method refactorings are presented in Table 4. From the table, we make the following observations:

Table 4 Results of classification

Application	$Type_1$	$Type_2$	$Type_{3A}$	$Type_{3B}$	$Type_3$
FindBugs/%	31	19	29	21	50
Hibernate/%	42	14	35	9	44
JEdit/%	25	16	41	17	58
JFreeChart/%	45	8	34	14	48
Weka/%	23	10	57	10	67
JasperReports/%	13	29	32	26	58
Vuze-Azureus/%	22	14	53	10	64
Total/%	28	16	40	16	56

- Firstly, current reuse is the primary motivation for more than half of the extract method refactorings ($Type_3$). From the table, we observe that $Type_3$ accounts for 44%–67% of all extract method refactorings (56% on average). On all of the subject applications, $Type_3$ accounts for the largest portion. The results suggest that current reuse is the predominant motivation for extract method refactorings, which is consistent with interview results in Section 3.
- Secondly, decomposition of methods is the primary mo-

tivation for around one third of the extract method refactorings ($Type_1$). From the table, we observe that $Type_1$ accounts for 13%–45% of extract method refactorings, and on average it accounts for 28%.

- Finally, clone resolution is the primary motivation for 16% of the extract method refactorings ($Type_2$). From the table, we observe that $Type_2$ accounts for 8%–29% of extract method refactorings, and on average it accounts for 16%.

4.4.3 Future reuse of new methods (answering RQ2)

We investigated the chances for methods generated by extract method refactorings in version n to be reused in future, i.e., in versions $n+1$, $n+2$ and so on. Results are presented in Table 5.

Table 5 Reuse of new methods generated by extract method refactorings

Application	*ALL*	$Type_1$	$Type_2$	$Type_{3A}$	$Type_{3B}$
FindBugs/%	12	3	23	15	12
Hibernate/%	6	6	4	2	23
JasperReports/%	15	11	15	8	26
JEdit/%	20	13	19	13	48
JFreeChart/%	29	10	26	38	67
Vuze-Azureus/%	13	5	0	14	39
Weka/%	16	15	13	10	57
Total/%	17	10	15	15	39

From the table, we observe that most (83%) of such methods are not reused in future although all the involved applications have an evolutionary history longer than three years (8.9 years on average). It might suggest that to extract fragments as new methods in order to facilitate future reuse (instead of current reuse) might not be very cost-effective.

From Table 5, we also observe that methods generated by $Type_{3B}$ extract method refactorings (to facilitate current reuse and to resolve clones at the same time) are much more likely to be reused in future than those generated by other extract method refactorings. On average, 39% of such methods have been reused in later versions whereas only 10%, 15%, and 15% of the methods generated by $Type_1$, $Type_2$, and $Type_{3A}$ extract method refactorings respectively have been reused in later versions.

We also investigated chances for methods generated by extract method refactorings in version n to be reused in the next version, i.e., version $n+1$. Only 1.8%, 3.9%, 3.8%, and 7.4% of the new methods generated by $Type_1$, $Type_2$, $Type_{3A}$ and $Type_{3B}$ extract method refactorings respectively have been reused in the next version. On average, 3.9% of such methods hare been reused in the next version.

From the results presented in preceding paragraphs, we might make the following observation:

- If the primary motivation for extract method refactorings is method decomposition ($Type_1$) or clone resolution ($Type_2$), the resulting new methods have a small chance (12% on average) to be reused in future.
- If the extracted fragment could be reused immediately in software evolution and extracting such a fragment can resolve existing clones ($Type_{3B}$), the extracted method has a much higher chance (around 40%) to be reused in future.

We conclude from these results that it is rarely gainful to extract methods only for future reuse. Considering that most of the involved developers (63%) take future reuse as one of the major motivations for extract method refactoring (as suggested in Section 3), these findings are surprising. While the findings and their supporting data are presented to the developers who list future reuse as a major motivation in the interview, all of them are surprised. Most of them (73%=11/15) declared that they would think twice on conducting such refactorings in future.

5 Threats to validity

A threat to external validity is that only 25 developers were interviewed. Interview is usually time-consuming, and thus it is difficult to interview a large number of busy developers. To reduce the threat, we interviewed students with previous industry experiences besides those from the industry.

The second threat to external validity is that only seven applications were involved in the refactoring history based validation. Such applications were developed by different developers and were in different domains, which might help to reduce the threat. It is also possible to involve more applications in future because most of the analysis involved in the study is automated.

The third threat to external validity is that the refactoring history based validation is limited to Java applications. Because the tool used to discover extract method refactorings, i.e., Extract Method Detector, works for Java applications only, and all of the selected subject applications are in Java. However, conclusions drawn on Java applications might not hold for applications in other languages, e.g., C++ and C♯.

A threat to internal validity is that some extract method refactorings might be missed by Extract Method Detector. Some extract method refactorings might be conducted, and

then related source code is removed for reasons such as changes of functionality. As a result, such refactorings could not be identified anymore.

Another threat to internal validity is that the classification of extract method refactorings might be inaccurate in some cases. Firstly, the classification approach presented in Section 4.4 takes the assumption that none of the callers of the extracted method was removed. However, the assumption might not hold in some cases. For example, if an extract method refactoring was conducted to resolve clones in methods m_1 and m_2, these methods should call the extracted new method m. However, if one of the callers, e.g., m_1 was removed for some reasons, the classification approach would misinterpret the refactoring as a $Type_1$ extract method refactoring because there was only a single caller of method m. Secondly, the classification approach was designed to distinguish between three major application scenarios of extract method refactorings, i.e., method decomposition, clone removal, and current reuse. However, some other special application scenarios might exist as well. To improve the recall of refactoring identification, the proposed approach does not require new methods to exactly match the extracted (removed) source code because it is common to modify such code after extraction to pass messages between the new method and other methods or to generalize the source code for reuse. However, it is also possible that the new code (or modified code) in the new method is the result of bug fixing, or implementing a change in a requirement. In that case, the motivation could be interpreted in a different way. For example, the developer may extract a method to facilitate bug fixing. The motivation in that case, is not strictly code reuse, or method decomposition, or clone elimination, but rather facilitating a maintenance task (bug fix, feature addition, or requirement change). Adding a special category for this kind of motivations, e.g., *facilitate functionality extension* [13], may increase the accuracy of the classification. Finally, for the three major application scenarios, we did not make further classification. For example, the first scenario (to decompose methods) might be further classified into the following sub-scenarios: decomposition of long methods, decomposition of complex methods, and decomposition of methods with low cohesion.

Another threat to internal validity is that the reuse of extracted methods might be misinterpreted. In the study, we checked whether the extracted methods had been reused in the ensuing versions that we extracted. However, methods that have not yet been reused might be reused in the future since such applications might evolve in the future. To reduce the threat, we selected subject applications with long evolution histories.

6 Conclusions and future work

To have a better understanding of extract method refactorings, in this paper we investigated why developers conduct extract method refactorings. By interviewing developers, we discovered the major motivations for extract method refactorings. We also validated the interview results on open-source applications. Our results suggest that current reuse, decomposition of long methods, clone resolution, and future reuse are the major motivations for extract method refactorings. Our results also suggest that current reuse is the primary motivation for 56% of extract method refactorings, decomposition of methods is the primary motivation for 28% of extract method refactorings, and clone resolution is the primary motivation for 16% of extract method refactorings.

We also investigated the likelihood for the extracted methods to be reused in future, and our results suggest that such methods have a small chance (12%) to be reused in future unless the extracted fragments are reused immediately in software evolution and extracting such a fragments resolves existing clones at the same time.

Our findings are interesting to developers providing tool supporting for extract method refactorings in the following aspects. Firstly, our findings suggest that recommending extract method opportunities by analyzing inner structure, e.g., complexity and length of methods alone might miss many extract method opportunities. Secondly, our findings also suggest that extract method refactorings are often driven by current and immediate reuse. Consequently, how to recognize or predict reuse requirements timely during software evolution may play a key role in the recommendation and automation of extract method refactorings.

Our findings are interesting to practitioners of extract method refactorings as well. Our results suggest that on average only 17% of new methods generated by extract method refactorings are reused in future. Consequently, practitioners of extract method refactorings should think twice on the costs and benefits of such refactorings if future reuse is one of the motivations (and expected benefits) for such refactorings.

In future, to make the study more convincing, we would like to interview more developers and analyze more applications. It is also interesting to analyze extract method refactorings collected in other ways, e.g., recorded by IDEs. Murphy-Hill et al. [44] have compared four approaches to collecting refactoring histories, and they found that each of them has

its own advantages and disadvantages. Consequently, analysis of extract method refactorings collected in different ways might make the conclusions more convincing and robust.

Acknowledgements The work was funded by the National Natural Science Foundation of China (Grant Nos. 61272169, 61472034), Program for New Century Excellent Talents in University (NCET-13-0041), and Beijing Higher Education Young Elite Teacher Project (YETP1183).

References

1. Opdyke W F. Refactoring object-oriented frameworks. Dissertation for the Doctoral Degree. Champaign: University of Illinois at Urbana-Champaign, 1992
2. Mens T, Tourwé T. A survey of software refactoring. IEEE Transactions on Software Engineering, 2004, 30(2): 126–139
3. Murphy-Hill E, Parnin C, Black A P. How we refactor, and how we know it. IEEE Trransactions on Softeare Engineering, 2012, 38(1): 5–18
4. Xing Z, Stroulia E. Refactoring practice: how it is and how it should be supported — an eclipse case study. In: Proceedings of IEEE International Conference on Software Maintenance. 2006, 458–468
5. Maruyama K. Automated method-extraction refactoring by using block-based slicing. ACM SIGSOFT Software Engineering Notes, 2001, 26(3): 31–40
6. Sharma T. Identifying extract-method refactoring candidates automatically. In: Proceedings of the 5th Workshop on Refactoring Tools. 2012, 50–53
7. Tsantalis N, Chatzigeorgiou A. Identification of extract method refactoring opportunities for the decomposition of methods. Journal of Systems and Software, 2011, 84: 1757–1782
8. Silva D, Terra R, Valente M T. Recommending automated extract method refactorings. In: Proceedings of the 22nd International Conference on Program Comprehension. 2014, 146–156
9. Palomba F, Bavota G, Di Penta M, Oliveto R, De Lucia A. Do they really smell bad? a study on developers' perception of bad code smells. In: Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution. 2014, 101–110
10. Murphy-Hill E, Parnin C, Black A P. How we refactor, and how we know it. In: Proceedings of the 31st International Conference on Software Engineering. 2009, 287–297
11. Liu H, Guo X, Shao W Z. Monitor-based instant software refactoring. IEEE Transactions on Software Engineering, 2013, 39(8): 1112–1126
12. Niu N, Bhowmik T, Liu H, Niu Z. Traceability-enabled refactoring for managing just-in-time requirements. In: Proceedings of the 22nd IEEE International Requirements Engineering Conference. 2014, 133–142
13. Tsantalis N, Guana V, Stroulia E, Hindle A. A multidimensional empirical study on refactoring activity. In: Proceedings of the 2013 Conference of the Center for Advanced Studies on Collaborative Research. 2013, 132–146
14. Fowler M. Refactoring: improving the design of existing code. In: Wells D, Williams L, eds. Extreme Programming and Agile Methods — XP/Agile Universe 2002. 2002
15. Murphy G C, Kersten M, Findlater L. How are Java software developers using the eclipse IDE? IEEE Software, 2006, 23(4): 76–83
16. Kim M, Zimmermann T, Nagappan N. A field study of refactoring challenges and benefits. In: Proceedings of the 20th ACM SIGSOFT International Symposium on the Foundations of Software Engineering. 2012, 1–11
17. Bavota G, Lucia A D, Penta M D, Oliveto R, Palomba F. An experimental investigation on the innate relationship between quality and refactoring. Journal of Systems and Software, 2015, 107: 1–14
18. Palomba F, Bavota G, Penta M D, Oliveto R, Poshyvanyk D, Lucia A D. Mining version histories for detecting code smells. IEEE Transactions on Software Engineering, 2015, 41(5): 462–489
19. Palomba F. Textual analysis for code smell detection. In: Proceedings of the 37th IEEE/ACM International Conference on Software Engineering. 2015, 769–771
20. Bavota G, Oliveto R, Gethers M, Poshyvanyk D, De Lucia A. Methodbook: recommending move method refactorings via relational topic models. IEEE Transactions on Software Engineering, 2014, 40(7): 671–694
21. Bavota G, Gethers M, Oliveto R, Poshyvanyk D, de Lucia A. Improving software modularization via automated analysis of latent topics and dependencies. ACM Transactions on Software Engineering & Methodology, 2014, 23(1)
22. Bavota G, De Lucia A, Marcus A, Oliveto R. Automating extract class refactoring: an improved method and its evaluation. Empirical Software Engineering, 2014, 19(6): 1617–1664
23. Demeyer S, Ducasse S, Nierstrasz O. Finding refactorings via change metrics. In: Proceedings of the 15th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications. 2000, 166–177
24. Xing Z, Stroulia E. UMIDiff: an algorithm for object-oriented design differencing. In: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering. 2005, 54–65
25. Xing Z, Stroulia E. Refactoring detection based on UMLDiff change-facts queries. In: Proceedings of the 13th Working Conference on Reverse Engineering. 2006, 263–274
26. Weissgerber P, Diehl S. Identifying refactorings from source-code changes. In: Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering. 2006, 231–240
27. Dig D, Comertoglu C, Marinov D, Johnson R. Automated detection of refactorings in evolving components. In: Proceedings of the 20th European Conference on Object-Oriented Programming. 2006, 404–428
28. Broder A Z. On the resemblance and containment of documents. In: Proceedings of the Compression and Complexity of Sequences. 1997, 21–29
29. Prete K, Rachatasumrit N, Sudan N, Kim M. Template-based reconstruction of complex refactorings. In: Proceedings of the 2010 IEEE International Conference on Software Maintenance. 2010, 1–10
30. Godfrey M, Zou L. Using origin analysis to detect merging and splitting of source code entities. IEEE Transactions on Software Engineering, 2005, 31(2): 166–181
31. Kim S, Pan K, Whitehead E. When functions change their names: automatic detection of origin relationships. In: Proceedings of the 12th Working Conference on Reverse Engineering. 2005, 143–152
32. Malpohl G, Hunt J, Tichy W. Renaming detection. In: Proceedings of the 15th IEEE International Conference on Automated Software Engi-

neering. 2000, 73–80

33. Fluri B, Wursch M, Pinzger M, Gall H. Change distilling: tree differencing for fine-grained source code change extraction. IEEE Transactions on Software Engineering, 2007, 33(11): 725 –743
34. Kim M, Notkin D, Grossman D. Automatic inference of structural changes for matching across program versions. In: Proceedings of the 29th International Conference on Software Engineering. 2007, 333–343
35. Koschke R. Survey of research on software clones. In: Koschke R, Merlo E, Walenstein A, eds. Duplication, Redundancy, and Similarity in Software. 2007
36. Wang T, Harman M, Jia Y, Krinke J. Searching for better configurations: a rigorous approach to clone evaluation. In: Proceedings of the 9th Joint Meeting on Foundations of Software Engineering. 2013, 455–465
37. Roy C K, Cordy J R, Koschke R. Comparison and evaluation of code clone detection techniques and tools: a qualitative approach. Science of Computer Programming, 2009, 74: 470–495
38. Jia Y, Binkley D, Harman M, Krinke J, Matsushita M. KClone: a proposed approach to fast precise code clone detection. In: Proceedings of the 3rd International Workshop on Software Clones. 2009
39. Balazinska M, Merlo E, Dagenais M, Lague B, Kontogiannis K. Advanced clone-analysis to support object-oriented system refactoring. In: Proceedings of the 7th Working Conference on Reverse Engineering. 2000, 98–107
40. Baxter I D, Yahin A, Moura L, Anna S M, Bier L. Clone detection using abstract syntax trees. In: Proceedings of the International Conference on Software Maintenance. 1998, 368–377
41. Canfora G, Cerulo L, Di Penta M. Ldiff: an enhanced line differencing tool. In: Proceedings of the 31st IEEE International Conference on Software Engineering. 2009, 595–598
42. Asaduzzaman M, Roy C, Schneider K, Di Penta M. Lhdiff: tracking source code lines to support software maintenance activities. In: Proceedings of the 29th IEEE International Conference on Software Maintenance. 2013, 484–487
43. Liu Y, Liu H. Automated detection of extract method refactorings. Technical Report. 2014
44. Murphy-Hill E, Black A P, Dig D, Parnin C. Gathering refactoring data: a comparison of four methods. In: Proceedings of the 2nd Workshop on Refactoring Tools. 2008, 1–5

Wenmei Liu received the BS degree in control science from Shandong University, China in 2003, and the MS degree from Beihang University, China in 2006. She is currently working toward the PhD degree with the School of Computer Science and Technology at the Beijing Institute of Technology, China. Her research interests include software engineering, software refactoring, and software evolution.

Hui Liu received the BS degree in control science from Shandong University, China in 2001, the MS degree in computer science from Shanghai University, China in 2004, and the PhD degree in computer science from Peking University, China in 2008. He is currently an associate professor in the School of Computer Science and Technology at the Beijing Institute of Technology, China. He is particularly interested in software refactoring, design pattern, and software evolution. He is currently doing research to make software refactoring easier and safer. He is also interested in developing practical refactoring tools to assist software engineers.