IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received October 24, 2018, accepted November 16, 2018, date of publication November 29, 2018,
date of current version December 27, 2018.

Digital Object Identifier 10.1109/ACCESS.2018.2883769

Dynamic Ranking of Refactoring Menu Items for Integrated Development Environment

THIDA OO, HUI LIU, AND BRIDGET NYIRONGO
School of Computer Science and Technology, Beijing Institute of Technology, Beijing 100081, China

Corresponding author: Hui Liu (liuhui08@bit.edu.cn)

This work was supported in part by the National Natural Science Foundation of China under Grant 61772071, Grant 61472034, and Grant 61690205, and in part by the National Key Research and Development Program of China under Grant 2016YFB1000801.

ABSTRACT Software refactoring is popular and thus most mainstream IDEs, e.g., Eclipse, provide a top level menu, especially for refactoring activities. The refactoring menu is designed to facilitate refactorings, and it has become one of the most commonly used menus. However, to support a large number of refactoring types, the refactoring menu contains a long list of menu items. As a result, it is tedious to select the intended menu item from the lengthy menu. To facilitate the menu selection, in this paper, we propose an approach to dynamic ranking of refactoring menu items for IDE. We put the most likely refactoring menu item on the top of the refactoring menu according to developers' source code selection and code smells associated with the selected source code. The ranking is dynamic because it changes frequently according to the context. First, we collect the refactoring history of the open source applications and detect the code smells. Based on the refactoring history, we design questionnaires and analyze the responses from developers to discover the source code selection patterns for different refactoring types. Subsequently, we analyze the relationship between code smells associated with the refactoring software entities and the corresponding refactoring types. Finally, based on the preceding analysis, we calculate the likelihood of different refactoring types to be applied when a specific part of source code is selected, and rank the menu items according to the resulting likelihood. We conduct a case study to evaluate the proposed approach. Evaluation results suggest that the proposed approach is accurate, and in most cases (95.69%), it can put the intended refactoring menu item on the top of the menu.

INDEX TERMS Software development, software refactoring, menu ranking, IDE.

I. INTRODUCTION

Software refactoring is a well-known technique that is widely adopted by software engineers to improve the design and enable the evolution of a system [1]. Software refactoring changes the internal structure of software systems without changing their external behaviors. The major purpose of software refactoring is to make source code more reusable, maintainable, extensible, and understandable [2]. Most of the modern integrated development environments (IDEs), e.g., Eclipse (http://eclipse.org), Visual Studio (http://microsoft.com/visualstudio) and IntelliJ IDEA (http://www.jetbrains.com/idea) provide tools support for software refactoring. For example, Eclipse has a top-level menu item specially designed for software refactoring. It provides entries for dozens of software refactorings that are automated or semi-automated by Eclipse. Tool support is crucial for the success of software refactoring. Eclipse has made 27 refactor actions as different types of refactorings are being added to it occasionally. With such 27 refactor actions menu, it is often tedious for software engineers to select the intended menu item.

For example, if developers want to go to Push Down method, which moves a method from a class to a specific subclass, the refactoring menu of Eclipse will appear as shown in Figure 1. Whereby, developer would find that name in a menu to apply the Push Down method refactoring. This process of choosing a menu item, appears to be tedious for developers.

Therefore, we present the dynamic ranking of refactoring menu items in IDE that the most likely refactoring menu item is placed on the top of refactoring menu according to source code selection and code smell associated with the selected source code. As a result, a developer does not need to go to the lengthy menu. The ranking is dynamic because it changes

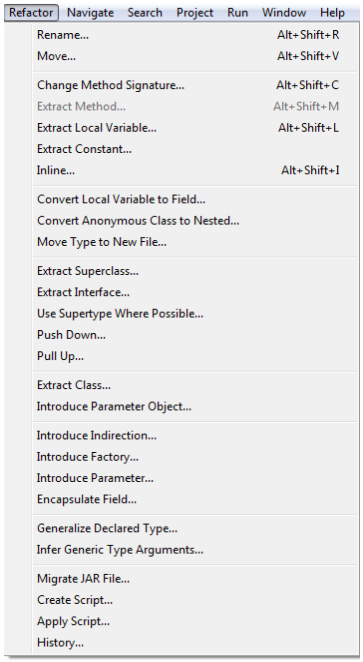


FIGURE 1. Refactoring menu of eclipse.

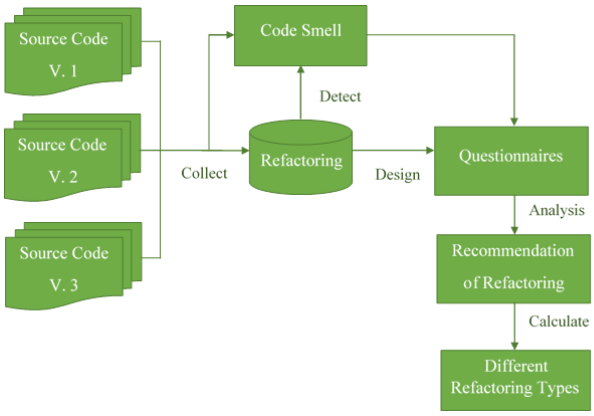


FIGURE 2. Overview of the proposed approach.

TABLE 1. Subject applications.

Application	Domain	Number of Versions	Size(LOC)
Hibernate	Java Persistence	31	68 929-176 879
Phex	File Sharing	11	37 489-249 691
Weka	Machine Learning	38	38 890-272 212

frequently according to the context. However, if we change the whole menu frequently (according to their likelihood to be selected), it could be difficult for developers to select the intended menu item when it is not on the top. Consequently, we split the whole menu into two parts: the top 1 (changing menu item, noted as T_1) and the others (fixed items, noted as OT). OT is identical to the refactoring menu of Eclipse, and all items are fixed. Our approach only changes the top 1 menu item (T_1) dynamically. When the ranking result is not correct (i.e., the top 1 item is not the intended one), developers have to select the intended menu item from OT that contains all refactoring actions. The theoretical foundation for such split menus could be found in [3].

The major contribution of this paper includes:

- An approach of dynamically ranking refactoring menu items for IDE based on source code element and code smells.
- The results showed that the proposed approach is accurate.

The rest of the paper is structured as follows. Section II illustrates the overview of the research method used in this paper. Section III presents an evaluation of the proposed approach on open-source applications. Section IV outlines some of the limitations to this research. Section V discusses related issues. Section VI provides conclusions and potential future work.

II. METHODOLOGY

A. OVERVIEW

An overview of the proposed approach is presented in Figure 2. As suggested by the figure, the proposed approach works as follows: First, we collect the refactoring history of the open source applications and detect the code smells. Based on the refactoring history, we design questionnaires and analyze the responses from developers to discover the source code selection pattern for different refactoring types. And then, we analyze the relationship between code smells associated with the refactored software entities and the corresponding refactoring types. Finally, we calculate the likelihood of different refactoring types and rank the menu items according to the resulting likelihood.

B. SUBJECT APPLICATIONS

An overview of the subject applications is presented in Table 1. These applications are as follows:

- Hibernate [4] is an open-source application developed in Red Hat. 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 (http://sourceforge.net/projects/hibernate). We analyzed the last 31 versions (from version 4.0.0 to version 5.6.1). The size of this application varies from 68,929 to 176,879 LOC.
- Phex [5] is a peer to peer file sharing client for the gnutella network. It is free software and distributed under the GNU General Public License (GPL). Source code of this application was downloaded from SourceForge (http://sourceforge.net/projects/phex). We analyzed 11 versions (from version 2.8.4.93 to version 3.4.2.116). The size of this application ranges from 37,489 to 249,691 LOC.
- Weka [6] is an open-source application developed by the machine learning group at the University

of Waikato. It implements in Java a set of well-known machine learning algorithms. Source code of this application was downloaded from its SVN server (http://svn.cms.waikato.ac.nz/svn/weka). We analyzed the last 38 versions (from version 3.1.7 to version 3.7.11). The size of this application ranges from 38,890 to 272,212 LOC.

We selected such subjects because of the following reasons. First, all of them are open-source applications and their source code is publicly available. Second, these applications were developed by different developers. Third, these applications are well-known and popular. Thus, it brings some sort of diversity on the different styles of coding and refactoring. And then, it would limit and bridge the gap to the external threats which might be available. Notably, we also consider the domains (to cover different domains) while selecting such applications.

C. DISCOVERY OF REFACTORING

In order to collect the refactoring histories for the dynamic ranking of refactoring menu items, we leveraged refactoring recovery tools to discover the evolutionary history of software application.

First, we employed Refactoring Crawler [7], a well-known refactoring discovery tool, to detect a set of potential previously applied refactorings. The tool uses a combination of a fast syntactic analysis to detect candidates and a more expensive semantic analysis to refine the results.

Second, we also applied Logical Structural Diff (LSDiff), another well-known refactoring discovery tool, to the subject applications and it generates a set of potential refactorings as well. LSDiff makes it easier for developers to understand code changes by grouping related differences as a single rule and by finding exceptions that indicate missed or inconsistent updates [8]. Third, we merge these two data sets and manually check each of the potential refactorings and associated code smells with the program code. The process of collecting the various refactoring histories involved the following three basic steps:

- Importing the various versions of the chosen software subjects into eclipse.
- Selecting the subjects that represented two successive version of the software for refactoring discovery by using Refactoring Crawler and LSDiff.
- Checking whether the discovered refactoring was an actual refactoring or not.

We discovered 279, 189 and 143 refactorings from Hibernate, Phex and Weka, respectively. In total, 611 refactorings were discovered from these subject applications as shown in Table 2. Table 3 shows the total number of each type of refactoring that were collected from the subject applications.

In Table 4, we identified and analyzed 611 refactorings classified in 17 commonly used refactoring types. 14 code smell types are used to classify the collected refactorings.

TABLE 2. Discovered refactorings (by subject applications).

Application	Number of Refactorings
Hibernate	279
Phex	189
Weka	143
Total	611

TABLE 3. Discovered refactorings (by refactoring type).

Refactoring	Number
Remove parameter	113
Change parameter type	85
Change method parameter name	46
Add parameter	70
Extract method	51
Extract interface	43
Inline temp	44
Inline method	37
Rename Method	45
Rename Package	24
Rename Class	4
Move Method	24
Push down Method	8
Pull up field	6
Pull up Method	5
Reorder parameter	4
Extract super class	2
Overall Total	**611**

These code smell types are selected because they are conceptually associated with the definition of the refactoring types that is each refactoring type is explicitly associated with one or more code smells. Code smells have been associated with refactorings when they were proposed by the experts who defined such code smells [9].

For each discovered refactoring, we manually check whether the refactored software entity is associated with code smells. We associated the identified code smell with the discovered refactoring if the refactoring is one of the solutions for the code smells [9]. Notably, researchers have proposed a number of automatic tools to identify such code smells, e.g., InsRefactor [10] and JDeodorant [11]. We do not use these tools in this paper because of the following reasons. First, such tools may miss some code smells or report false negatives. Second, we want to keep the paper simple and focused, excluding the impact of smell detection tools on the proposed approach. However, such tools could be employed in practice because it could be time consuming and even impractical if developers have to manually identify code smells before refactoring menus could be ordered and shown up.

TABLE 4. Relation between code smells and refactoring number.

Code smell (C_j)	Refactoring type and frequency (r_i)	P(r_i/C_j)
Feature envy	Move method/7	100%
Inappropriate intimacy	Move method/4	100%
Shot gun surgery	Move method/2	100%
Refused bequest	Push down method/3	100%
Dead code	Remove parameter/76	100%
Unnecessary variable	Inline temp/24	100%
Large class	Extract interface/23	100%
Middle man	Inline method/13	100%
Speculative generality	Inline method/23	79.31%
	Rename method/6	20.69%
Duplicate code	Extract method/19	76%
	Pull up method/3	12%
	Pull up field/2	8%
	Extract super class/1	4%
Alternative classes with different interfaces	Add parameter/35	74.47%
	Rename method/8	17.02%
	Move method/3	6.38%
	Extract super class/1	2.13%
Data class	Extract method/12	70.59%
	Move method/5	29.41%
Long method	Extract method/17	56.67%
	Inline temp/13	43.33%
Comments	Rename method/22	53.66%
	Rename package/16	39.02%
	Rename class/3	7.32%
No smell	Change method parameter/85	31.48%
	Change method parameter/46	17.04%
	Remove parameter/37	13.70%
	Add parameter/35	12.96%
	Extract interface/20	7.41%
	Rename method/9	3.33%
	Rename package/8	2.96%
	Inline temp/7	2.59%
	Push down method/5	1.85%
	Pull up field/4	1.48%
	Reorder parameter/4	1.48%
	Extract method/3	1.11%
	Move method/3	1.11%
	Pull up method/2	0.74%
	Inline method/1	0.37%
	Inline method/1	0.37%

We present the likelihood of relation between code smells and refactoring number in Table 4. The likelihood P(r_i/C_j) for a specific refactoring type r_i to be selected when a specific code smell C_j is associated with the selected element.

- Let $M_n(C_j,r_i)$ be the total number of refactoring type r_i that are associated with code smell C_j and
- Let $M(C_j)$ be the total number of refactorings associated with C_j, then

The likelihood of P(r_i/C_j) is calculated as follows:

$$P(r_i/C_j) = M_n(C_j, r_i)/M(C_j) \tag{1}$$

As an example, we calculate the likelihood of Inline method and Rename method when Speculative generality is associated with the selected element. Since, there are two types of refactorings associated with Speculative generality then:

$$\begin{aligned} M_n &= M_1 + M_2 \\ &= 23 + 6 \\ &= 29 \end{aligned} \tag{2}$$

P(Inline method/Speculative generality)=(23/29)=79.31%
P(Rename method/Speculative generality)=(6/29) =20.69%

When the code smell of Speculative generality is 79.31% likelihood, that Inline method will be applied, and thus it is the most likely refactoring in this case. In Table 4, we observed that Move method refactoring is the one which is related to most of the available code smell than other refactorings.

D. QUESTIONNAIRES

In this section, we design questionnaires to investigate developers' source code element selection to carry out different kinds of refactorings. Each question concerns with a refactoring selected from the 611 refactorings discovered from the subject applications. A sample questionnaire is available online in https://github.com/liuhuigmail/FeatureEnvy/blob/master/Questio nnaire.pdf [12]. Before distributing the questionnaires, we consulted three developers who have a minimum of five years of general programming experience about potential source code selection patterns for different kinds of refactorings. This was done so as to provide the correct source code selection patterns in the questionnaires associated with a particular refactoring. Participants had 50 questions and there were different sets of questionnaires assigned to different respondents to make sure that all discovered refactorings are involved in the questionnaires. The purpose of this questionnaires is just to understand and get a clear picture of what kind of elements you would select as a developer to carry out a certain refactoring and to know what kind of code smells would be associated with that selected element for that specific refactoring.

- What kind of element would you select to rename the class?
- What kind of element would you select to rename the method?
- What kind of element would you select to pull up the method?
- What kind of element would you select to move the method?
- What kind of element would you select to change the method parameter?
- What kind of element would you select to push down the method?

- What kind of element would you select to rename the package?
- Which code smell would you make to select that element?

The respondents were supposed to indicate which element they would select to change the parameter type of the method in question. Their choice was to be indicated by providing the name of the element they would select against the question in cases where their element of choice was not included in the given options. The questionnaires were distributed to a group of 42 students studying in computer science and technology at a University. This group consists of Master and PhD students. All of the 42 students responses were valid and useful.

The results from the questionnaires were collected and analyzed as follows. First, we checked for each refactoring types which source code element the respondents would use to apply a particular refactoring. Second, all the refactorings which have the same source code element are grouped together. Third, for each of the refactorings with the same source code element, the likelihood of applying a particular refactoring was calculated when the type of source code is selected. That is, when that particular source code element is selected each refactoring is the intended one. These calculations were done as follows:

- Let $N_{i,j}(e_j,r_i)$ be the total number of respondents who selected the particular source code element of type e_j for a specific refactoring type r_i.
- Let $N_i(e_j)$ be the total number of respondents who selected the particular source code element of type e_j to apply refactorings.
- Let $P(r_i,e_j)$ be the likelihood of applying a particular refactoring type r_i while the source code element of type e_j is selected.

$$P(r_i, e_j) = N_{i,j}(e_j, r_i)/N_i(e_j) \tag{3}$$

As an example, when A set of methods are selected, we calculate the likelihood of Extract interface and Extract superclass. There are two refactoring types under which a developer would select A set of methods then:

$$\begin{aligned} N_i &= N_1 + N_2 \\ &= 17 + 6 \\ &= 23 \end{aligned} \tag{4}$$

P(Extract interface, A set of methods) = (17/23) = 73.91%
P(Extract superclass, A set of methods) = (6/23) = 26.09%

This means that when A set of methods are selected there is 73.91% likelihood that the intended refactoring to be applied is Extract interface. The rest of the results from the questionnaires are presented in Table 5.

From Table 5, we observe that Method name is the source code element associated with the greatest number of the refactorings. This makes it to be the main source code selection when applying a refactoring. Based on the questionnaire responses, we predict which refactorings should be ranked on top of a refactoring menu. The prediction is based on the selected source code element and the code smell. We only consider the selection patterns associated with a code smell. The refactorings included in the questionnaire had an association with a code smell from the initial discovery of refactorings. The calculations were carried out as follows:

- Let $N(e_j,C_j,r_i)$ be the total number of selection source code element of type e_j when the source code is

TABLE 5. Relation between selected element and intended refactoring.

Selected element (e_j)	Refactoring (r_i)	$P(r_i, e_j)$
Package name	Rename package	100%
Statements to be extracted	Extract method	100%
Type of method parameters	Change method parameter type	100%
Variable	Inline temp	100%
Method declaration	Inline temp	100%
The call site of the method	Inline method	100%
A set of methods	Extract interface	73.91%
	Extract superclass	26.09%
Field	Pull up field	66.67%
	Inline temp	33.33%
Method parameters	Add parameter	60%
	Change method parameter name	53.08%
	Reorder parameter	40%
	Remove parameter	31.75%
	Change method parameter type	8.53%
	Reorder parameter	6.64%
Body of method	Move method	47.19%
	Push down method	21.65%
	Pull up method	14.29%
	Inline method	10.82%
	Pull up field	6.05%
Body of class	Rename class	40.20%
	Extract interface	25.51%
	Extract superclass	23.51%
	Pull up field	10.78%
Class name	Rename class	34.99%
	Push down method	33.61%
	Pull up method	23.42%
	Move method	4.68%
	Extract superclass	3.30%
Method name	Rename method	30.26%
	Push down method	28.03%
	Move method	21.54%
	Pull up method	15.73%
	Remove parameter	2.91%
	Extract method	1.20%
	Change method parameter name	0.33%

associated with code smells of type C_j and the applied refactoring type r_i.

- Let $N(e_j,C_j)$ be the total number of selection source code element of type e_j that are associated with code smells of type C_j.

Then the likelihood of a refactoring r_i to be ranked on top of the refactoring menu will be given by:

$$P(r_i, e_j, C_j) = N(e_j, C_j, r_i)/N(e_j, C_j) \quad (5)$$

As an example, we consider the selection pattern of Package name which is associated with comments smell:

- There are 16 comments smells associated with package name, then $N(e_j,C_j) = 16$
- There is only Rename Package refactoring associated with this selection, then $N(e_j, C_j, r_i) = 16$

The likelihood of Rename Package refactoring to be ranked on top of the refactoring menu will be given by: P(Rename package,Package name,comment)=(16/16)=100%

Since there is no other refactoring which has a Package name source code selection associated with comments, it means upon such a selection Rename package refactoring will be the most likely refactoring to be on the refactoring menu. If there were more refactorings associated with such, the menu will be ranked according to which refactoring has the highest likelihood. Table 6 shows all of such refactoring menu rankings.

III. EVALUATION

A. RESEARCH QUESTIONS

In this section, we investigate the following questions:

- RQ1: How often are the intended refactoring menu items ranked on the top of the menu by the proposed approach?
- RQ2: How important are code smells and source code elements selections in the dynamic ranking of refactoring menu items IDE?

RQ1 explores the accuracy of the proposed approach in ranking the correct refactoring which a developer would want to carry out at a particular point in time. RQ2 explores which element between code smell and source code element is the best to use when ranking and recommending refactorings on a refactoring menu. These research questions are important because the proposed approach will not be useful if it recommends refactorings which a developer does not want to carry out at a particular point in time. Investigating these questions would reveal whether or not the proposed approach is significant.

B. SET UP

Apache Derby [13] and FindBugs [14] were chosen as testing data sets for our approach and details are presented as follows:

- Apache Derby [13] is an Apache DB subproject sponsored by Apache Software Foundation. It is an open-source relational database implemented entirely in Java.

TABLE 6. Top refactoring for different selections.

Element (e_k)	Smell(C_j)	Most likely Refactoring $P(r_i, e_j, C_j)$	Likelihood ($N(e_j,C_j,r_i)$ /$N(e_j,C_j)$)
Package name	comments	Rename package	100%
Statements to be extracted	Data class	Extract method	100%
Statements to be extracted	Duplicate code	Extract method	100%
Statements to be extracted	Long method	Extract method	100%
Variable	Unnecessary variable	Inline temp	100%
Variable	Long method	Inline temp	100%
Method declaration	Unnecessary variable	Inline temp	100%
Method declaration	Long method	Inline temp	100%
The call site of a method	Speculative generality	Inline method	100%
The call site of a method	Middle man	Inline method	100%
A set of methods	Alternative classes with different interfaces	Extract superclass	100%
A set of methods	Duplicate code	Extract superclass	100%
A set of methods	Large class	Extract interface	100%
Field	Unnecessary variable	Inline temp	100%
Field	Long method	Inline temp	100%
Field	Duplicate code	Pull up field	100%
Method parameter	Dead code	Remove parameter	100%
Method parameters	Alternative classes with different interfaces	Add parameter	100%
Body of method	Alternative classes with different interfaces	Move method	100%
Body of method	Feature envy	Move method	100%
Body of method	Data class	Move method	100%
Body of method	Inappropriate intimacy	Move method	100%
Body of method	Shot gun surgery	Move method	100%
Body of method	Duplicate code	Pull up method	100%
Body of method	Refused bequest	Push down method	100%
Body of method	Speculative generality	Inline method	100%
Body of method	Middle man	Inline method	100%
Body of class	Comments	Rename class	100%
Body of class	Large class	Extract interface	100%
Body of class	Alternative classes with different interfaces	Extract superclass	100%
Body of class	Duplicate code	Pull up field	60.00%
		Extract superclass	40.00%

TABLE 6. *Continued.* **Top refactoring for different selections.**

Element (e_k)	Smell(C_j)	Most likely Refactoring $P(r_i, e_k, C_j)$	Likelihood ($N(e_k,C_j,r_i)$ /$N(e_k,C_j)$)
Class name	comments	Rename class	100%
Class name	Feature envy	Move method	100%
Class name	Data class	Move method	100%
Class name	Shot gun surgery	Move method	100%
Class name	Alternative classes with different interfaces	Move method	100%
Class name	Inappropriate intimacy	Move method	100%
Method name	Dead code	Remove parameter	100%
Method name	Comments	Rename method	100%
Method name	Speculative generality	Rename method	100%
Method name	Feature envy	Move method	100%
Method name	Inappropriate intimacy	Move method	100%
Method name	Shot gun Surgery	Move method	100%
Method name	Long method	Extract method	100%
Method name	Refused bequest	Push down method	100%
Method name	Alternative classes with different interfaces	Rename method	85.49%
		Move method	14.51%
Method name	Data class	Move method	83.82%
		Extract method	16.18%
Method name	Duplicate code	Pull up method	78.57%
		Extract method	21.43%

We analyzed 11 versions of this application (from version 10.6.1.0 to version 10.12.1.1). The size of Apache Derby varies from 258,295 to 626,560 LOC. The source files of this application was downloaded from (http://svn.apache.org/repos/asf/db/derby).

- FindBugs [14] is a bug detection tool. It uses static analysis to identify bugs in Java code. Source code of this application was downloaded from SourceForge. We analyzed the last 9 versions (from version 1.3.5 to version 3.0.0). The size of FindBugs varies from 81,563 to 123,259 LOC. The source files of this application was downloaded from (http://findbugs.sourceforge.net).

These applications were chosen for the following reasons. First, they are open source software and such the source code is readily available. Second, these applications were developed by different developers from the other applications used to come with the approach for the dynamic ranking of refactoring menu. Finally, Apache Derby and FindBugs applications have a long evolutionary history, it is highly possible that a good number of refactoring histories could be discovered.

TABLE 7. **Refactorings (derby and findbugs).**

Refactoring	Number
Extract method	68
Move Method	54
Rename Method	43
Add parameter	29
Remove parameter	25
Change method parameter name	21
Extract interface	17
Inline temp	17
Change parameter type	15
Pull up Method	14
Push down Method	8
Inline method	8
Reorder parameter	6
Pull up field	5
Rename Package	4
Extract superclass	3
Rename Class	2
Overall Total	**339**

The discovered refactorings shown in Table 7 were assigned to five years experiences Java developers who work independently to repeat the discovered refactorings. All of them have at least five years of software development experience. We assessed the accuracy of the recommendation by comparing the type of the refactoring and the recommendation of the refactoring menu. The recommendation is correct if and only if they match.

In total, 339 refactorings were discovered from Apache Derby and FindBugs, we therefore assessed to deduce how many of them will be recommended based on the source code element and code smell associated with the source code element according to the refactoring menu.

C. RESULTS

To calculate the accuracy in recommending the correct refactorings, we considered the total number of refactorings which will be carried out by each of these developers, that is, (5*339) which gave us about 1695 refactorings. Out of these 1695 refactorings, 1622 (95.69%) refactorings were recommended based on both code smell and source code element, 1419 (83.72%) refactorings were recommended based on the code smell and 1268 (74.80%) refactorings were recommended based on the source code element. The result of these recommendations is shown in Table 8, 9 and 10 respectively.

Table 8 suggests that if we recommend menu items based on both code smells and source code, on 1622 out of the 1695 cases our recommendation is correct. Consequently,

TABLE 8. Accuracy in recommendation (based on both code smells and source code).

Refactoring	Accuracy
Remove parameter	100%
Rename method	100%
Add parameter	100%
Rename package	100%
Inline temp	100%
Pull up field	100%
Inline method	100%
Pull up method	100%
Push down method	100%
Rename class	98%
Extract superclass	90%
Extract interface	88%
Move method	84%
Extract method	80%
Overall Accuracy	**= (1622 / 1695)= 95.69%**

TABLE 9. Accuracy in recommendation (based on code smell only).

Smell	Accuracy
Inappropriate intimacy	100%
Shot gun surgery	100%
Feature envy	100%
Refused bequest	100%
Dead code	100%
Unnecessary variable	100%
Large class	100%
Middle man	100%
Data class	83.33%
Long method	83.33%
Duplicate code	71.43%
Speculative generality	60%
Alternative classes with different interfaces	42.86%
Comments	33.33%
Overall Accuracy	**= (1419 / 1695)= 83.72%**

the accuracy of the recommendation is 1622 / 1695 = 95.69%. Moreover, Table 9 suggests that if we recommend menu items based on the code smell only, on 1419 out of the 1695 cases our recommendation is correct. The accuracy of the recommendation is 1419 / 1695 = 83.72%. Table 10 suggests that if we recommend menu items based on selected source code element only, on 1268 out of the 1695 cases our recommendation is correct. The accuracy of the recommendation is 1268 / 1695 = 74.80%. All such three approaches work on the same testing samples (1695 refactorings).

However, recommending refactorings can not only be based on code smells, because at times a developer would carry out a refactoring even if a code smell does not exist, in which case the source code selected would determine which kind of refactoring to apply. So, when ranking and recommending of refactoring menu, we would better consider both the code smell and source code element. Therefore, the approach for the dynamic ranking of a refactoring menu needs to use both the code smell and source code element.

IV. THREATS TO VALIDITY

The first threat to external validity is that only two applications were used to validate the accuracy of the approach. There is a need to employ a good number of applications to confirm the validity of the approach in other applications.

The second threat is that only 5 developers were used to see and discuss on the source code elements and code smells to be associated with a particular refactoring. This was due to time constraints on the other developers who could have taken part in these discussions. There is a need to involve a wide range of developers from different sectors to make an overall and complete validation of the approach.

The last threat to external validity is that the validation of the approach was based on the Refactoring menu in Eclipse. There is a need to consider other Refactoring menus in the different available IDEs that support refactoring; because how the refactoring menu is implemented in Eclipse could not be the same as how it is implemented in other available IDEs e.g. Visual Studio.

A threat to internal validity is that the discovered refactorings from Derby and FindBugs might be inaccurate. The refactoring detectors used in the evaluation i.e., Refactoring Crawler and LSDiff, might result in false positives and false negatives. Even though the discovered refactorings were checked manually to get the actual refactorings; this manual process is usually error prone.

V. RELATED WORK

This section reviews the related literature in three different areas. The first group of related papers focuses on menu item organization. The second group focuses on identification of refactoring opportunities. The third group focuses on detection of refactoring.

A. MENU ORGANIZATION

Vanderdonckt *et al.* [15] presented Cloud Menus, a new type of adaptive split menu in which predicted menu items are arranged in a circular word cloud superimposed on the static menu.

Murphy-Hill *et al.* [16] proposed a mapping from gestures to refactorings and an implementation of that mapping in the form of marking menus. This result suggested that programmers can conclude the gesture that will invoke the appropriate refactoring tools, even if they do not know the name of the refactoring. Heo *et al.* [17] presented MelodicTap, a novel hotkey technique utilizing fingering gestures for touchscreen tablets and described the design and implementation of MelodicTap and our findings gained from the

TABLE 10. Accuracy in recommendation (based on source code element only).

Refactoring	Accuracy
Rename class	100%
Rename package	100%
Add parameter	100%
Move method	83.33%
Extract method	83.33%
Rename method	80%
Change parameter type	80%
Inline temp	71.43%
Change method parameter name	66.67%
Pull up field	57.14%
Inline method	40%
Extract interface	33.33%
Overall Accuracy	**= (1268 / 1695)= 74.80%**

user study. MelodicTap have three advantages: the use of finger tap sequences allows users to access a large number of menu items, the user of finger-mapped buttons enables eye-free operation and the sequential traversal of the hierarchical menu helps users develop an expert skill.

Park *et al.* [3] investigated the usability of different adaptable and adaptive menu interfaces in a desktop environment. An adaptable menu and two different adaptive menus were implemented and evaluated. The two adaptive menus consist of an adaptive split menu that moves frequently used menu items to the top and an adaptive highlight menu that boldfaces frequently used menu items. The adaptive split menu, which dynamically changes the items in the top based on the recent selection history.

Sears and Shneiderman [18] developed and applied design guidelines for split menus. They compared with an alphabetic menu and a frequency-ordered menu, and found that the split menu was better than others in terms of performance and satisfaction when the frequently selected items were located in the middle or bottom of the menu. In this research, split menus were significantly faster than alphabetic menus and yielded significantly higher subjective preferences. A possible resolution to the continuing debate among cognitive theorists about predicting menu selection times is offered. Their study offered evidence that, at least when selecting items from pull-down menus, a logarithmic model applies to familiar (high-frequency) items, and a linear model to unfamiliar (low-frequency) items. The results of the controlled experiment demonstrated not only the time savings and higher preference ratings split menus create, but also the value of the proposed guideline for creating split menus. Inspired by their evaluation results, we create split menus for refactorings.

Kandari and Jain [19] analyzed the organization of parent and child menu items on a user interface and is an empirical study to determine the best positions of anchoring points between the parent and child menus. The result of these analysis aimed at the most effective use of the menu traversal and access time, the findings obtained from the current study proves highly effective and saving increases multi-hold as application usage increases.

Liu *et al.* [20] explored the effects of frequency distribution on average menu performance and individual item performance. The results showed that user's behavior is sensitive to different frequency distributions at both menu and item level. The most surprising result is that individual item selection time depends on not only its own frequency but also the frequency of other items in the menu.

B. IDENTIFICATION OF REFACTORING OPPORTUNITIES

Silva *et al.* [21] presented JExtract, a recommendation system based on structural similarity that identifies Extract Method refactoring opportunities that are directly automated by IDE-based refactoring tools. Refactoring recommendation approaches for extract class refactoring have also been proposed and presented by [22], [23], and [24].

Bavota *et al.* [22] proposed an Extract Class refactoring method based on graph theory that exploits structural and semantic relationships between methods. The proposed approach used a weighted graph to represent a class to be refactored, where each node represents a method of the class. The weight of an edge that connects two nodes (methods) is a measure of the structural and semantic relationship between two methods that contribute to class cohesion.

Bavota *et al.* [23] proposed a method for automating the Extract Class refactoring. The proposed approach analyzed (structural and semantic) relationships between the methods in a class to identify chains of strongly related methods. The identified method chains are used to define new classes with higher cohesion than the original class, while preserving the overall coupling between the new classes and the classes interacting with the original class.

Bavota *et al.* [24] proposed an approach based on game theory that recommend extract-class refactoring opportunities. The approach modeled a non-cooperative game where two players contend for the methods of the original class to build two new classes with higher cohesion than the original class. The results achieved in a preliminary evaluation supported the applicability and superiority of game theory. These researchers have employed the notions of graph theory [22], [23] and game theory [24] respectively to decompose large class.

Terra *et al.* [25] presented a recommmendation approach that suggests Move Method refactorings using the static dependencies established by methods. They also compared the recommendations provided by JMove, JDeodorant, Methodbook, and inCode in two open-source systems, and results suggest that JMove is more accurate [25].

Charalampidou *et al.* [26] introduced an approach (accompanied by a tool) that aims at identifying source code chunks that collaborate to provide a specific functionality, and proposed their extraction as separate methods. They proposed an

approach for identifying Extract Method opportunities in the source code of Long Methods (namely SEMI), and ranking them according to the benefit that they yield in terms of cohesion.

Fokaefs *et al.* [27] proposed a novel method to improve the design quality of an object-oriented system by applying Extract Class refactorings. This could produce meaningful and conceptually correct suggestions and extract classes that developers would recognize as meaningful concepts.

Chandran and Varghese [28] conducted a survey on various clustering techniques for identifying the extract class opportunities. The survey showed that there are several clustering approaches for the identification. Among the techniques reviewed, hierarchical clustering technique identifies better extract class opportunities for performing extract class refactoring than partitioned or any other clustering algorithms.

Tsantalis and Chatzigeorgiou [29] proposed a methodology for the identification of Move Method refactoring opportunities that constitute a way for solving many common Feature Envy bad smells.

All these contributions towards the identification of refactoring opportunities are all address the same problem, that is, to make it easy for a developer to identify a particular refactoring. This is different from this research in that an assumption is made that the developer already knows and is aware of the refactoring they need to apply. Our main focus is to make the application of these refactorings easy and less involving for the program by ranking the most likely refactoring on the top of a refactoring menu. This approach suggests which menu item should be used from the refactoring menu.

C. DETECTION OF REFACTORING

Dig *et al.* [7] proposed refactoring crawler to detect refactoring history by comparing Java source code of two successive versions of the same application. The algorithm used by refactoring crawler uses two techniques; syntactic analysis to detect refactoring candidates and semantic analysis to refine the results. Refactoring crawler mainly identifies refactorings that took place during component evolution.

Loh and Kim [8], proposed LSDiff, a program differencing technique that automatically identifies systematic structural differences as logic rules. This tool can be implemented as an Eclipse plug-in and provides a summary of systematic structural differences along with textual differences within an Eclipse integrated development environment. Developers would use program differencing tools to understand what changed between two versions while carrying out peer code reviews, resolving parallel edit conflicts or isolating failure inducing changes and also high level software changes such as refactorings could be detected using such a tool.

Khelladi *et al.* [30] presented AD-ROOM, a tool for an automatic detection of refactorings in Object Oriented Models(OOMs). AD-ROOM is designed to detect all applied refactorings during an OOM evolution and implemented as a plugin for the Eclipse IDE, a wide-spread development environment for software developers.

Tsantalis *et al.* [31] introduced RefactoringMiner that was later extend by Silva *et al.* [32] to mine refactorings in large scale in git repositories by analyzing the differences between the source code of two revisions. RefactoringMiner provides an API and can be used as an external library independently from an IDE. This tool is capable of identifying 14 high-level refactoring types: Package/Class/Method, Move Class/Method/Field, Pull Up Method/Field, Push Down Method/Field, Extract Method, Inline Method and Extract Superclass/Interface. Prete *et al.* [33] introduced RefFinder approach that identifies complex refactorings between two program versions using a template based refactoring reconstruction and expressed each refactoring type in terms of template logic rules and uses a logic programming engine to infer concrete refactoring instances.

All these tools could be used to mine and discover the refactorings which might have taken place within an application. But they cannot provide any way or technique on how to recommend refactorings or predict on how refactorings should be ranked dynamically on a refactoring menu which is the objective of this research.

VI. CONCLUSIONS AND FUTURE WORK

The main idea of this research is to put the most likely refactoring on top of the refactoring menu according to the developers' source code selection and code smell associated with the selected source code. It would make the refactoring process less wearisome and enjoyable for software developers and minimize the time for developers spend on choosing a refactoring from the lengthy refactorings menu items in the available various IDEs. We calculate the likelihood of different refactoring types to be applied when a specific part of source code is selected and rank the menu items according to the resulting likelihood. The evaluation results suggest that the approach is accurate, in most cases, it can put the intended refactoring menu item on the top of the menu.

In future, to make the study more convincing, we would like to analyze more applications, interview more developers with a much longer experience with the refactoring and compare speed and accuracy of refactoring menus.

In this paper, we have not evaluated the usefulness or the efficiency of the technique of the proposed approach. Researchers in [3] and [18] have proved empirically that accurate ranking of dynamic menu items is useful. Based on such research results, in this paper, we proved that the proposed approach for the dynamic ranking of refactoring menu items accurately, and thus it could be useful. In future, however, we should evalute the usefulness of the proposed approach empirically. The best way to evaluate it is to integrate the proposed technique to Eclipse IDE as an alternative menu: static or dynamic refactoring ranking may be selected by developer. The number of developers adopting the technique would be relevant to evaluate the impact of the proposed technique.

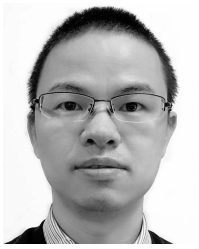
ACKNOWLEDGMENT

The authors would like to say thanks to the associate editor and the anonymous reviewers for their insightful comments and constructive suggestions.

REFERENCES

[1] D. Silva and M. T. Valente, "RefDiff: Detecting refactorings in version histories," in *Proc. IEEE/ACM 14th Int. Conf. Mining Softw. Repositories (MSR)*, May 2017, pp. 269–279.
[2] S. Hamioud and F. Atil, "Model-driven java code refactoring," *Comput. Sci. Inf. Syst.*, vol. 12, no. 2, pp. 375–403, 2015.
[3] J. Park, S. H. Han, Y. S. Park, and Y. Cho, "Usability of adaptable and adaptive menus," in *Proc. 2nd Int. Conf. Usability Internationalization*. Berlin, Germany: Springer-Verlag, 2007, pp. 405–411.
[4] *Hibernate*, Accessed: Apr. 1, 2018. [Online]. Available: https://hibernate.org/
[5] *Phex*. Accessed: Apr. 1, 2018. [Online]. Available: www.phex.org
[6] *Weka*. Accessed: Apr. 1, 2018. [Online]. Available: https://www.cs.waikato.ac.nz/ml/weka
[7] D. Dig, C. Comertoglu, D. Marinov, and R. Johnson, "Automated detection of refactorings in evolving components," in *Proc. Object-Oriented Programming (ECOOP)*. Berlin, Germany: Springer, 2006, pp. 404–428.
[8] A. Loh and M. Kim, "LSdiff: A program differencing tool to identify systematic structural differences," in *Proc. 32nd ACM/IEEE Int. Conf. Softw. Eng. (ICSE)*, vol. 2, New York, NY, USA, May 2010, pp. 263–266.
[9] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: XP/Agile Universe, 1999.
[10] H. Liu, X. Guo, and W. Shao, "Monitor-based instant software refactoring," *IEEE Trans. Softw. Eng.*, vol. 39, no. 8, pp. 1112–1126, Aug. 2013.
[11] M. Fokaefs, N. Tsantalis, and A. Chatzigeorgiou, "JDeodorant: Identification and removal of feature envy bad smells," in *Proc. 23rd IEEE Int. Conf. Softw. Maintenance (ICSM)*, Paris, France, Oct. 2007, pp. 519–520.
[12] *Questionnaire*. Accessed: Apr. 1, 2018. [Online]. Available: https://github.com/liuhuigmail/featureenvy/blob/master/questionnaire.pdf
[13] *Apache Derby*. Accessed: Apr. 1, 2018. [Online]. Available: https://db.apache.org/derby
[14] *Findbuds*. Accessed: Apr. 1, 2018. [Online]. Available: https://findbugs.sourceforge.net
[15] J. Vanderdonckt, S. Bouzit, G. Calvary, D. Chêne, "Cloud menus: A circular adaptive menu for small screens," in *Proc. 23rd Int. Conf. Intell. User Interfaces. (IUI)*, New York, NY, USA, 2018, pp. 317–328.
[16] E. Murphy-Hill, M. Ayazifar, and A. P. Black, "Restructuring software with gestures," in *Proc. IEEE Symp. Vis. Lang. Human-Centric Comput. (VL/HCC)*, Pittsburgh, PA, USA, Sep. 2011, pp. 165–172.
[17] S. Heo, J. Jung, and G. Lee, "MelodicTap: Fingering hotkey for touch tablets," in *Proc. 28th Austral. Conf. Comput.-Human Interact. (OzCHI)*, New York, NY, USA, 2016, pp. 396–400.
[18] A. Sears and B. Shneiderman, "Split menus: Effectively using selection frequency to organize menus," *ACM Trans. Comput.-Human Interact.*, vol. 1, no. 1, pp. 27–51, Mar. 1994.
[19] P. Kandari and A. Jain, "An empirical study to evaluate the best anchoring positions in nested menus for optimized access time," *SIGSOFT Softw. Eng. Notes*, vol. 40, no. 4, pp. 1–8, Jul. 2015.
[20] W. Liu, G. Bailly, and A. Howes, "Effects of frequency distribution on linear menu performance," in *Proc. CHI Conf. Human Factors Comput. Syst. (CHI)*, New York, NY, USA, 2017, pp. 1307–1312.
[21] D. Silva, R. Terra, and M. Valente. (Jun. 2015). "JExtract: An eclipse plug-in for recommending automated extract method refactorings." [Online]. Available: https://arxiv.org/abs/1506.06086
[22] G. Bavota, A. De Lucia, and R. Oliveto, "Identifying extract class refactoring opportunities using structural and semantic cohesion measures," *J. Syst. Softw.*, vol. 84, no. 3, pp. 397–414, Mar. 2011.
[23] G. Bavota, A. De Lucia, A. Marcus, and R. Oliveto, "Automating extract class refactoring: an improved method and its evaluation," *Empirical Softw. Eng.*, vol. 19, no. 6, pp. 1617–1664, Dec. 2014.
[24] G. Bavota, R. Oliveto, A. De Lucia, G. Antoniol, and Y.-G. Guéhéneuc, "Playing with refactoring: Identifying extract class opportunities through game theory," in *Proc. IEEE Int. Conf. Softw. Maintenance*, Sep. 2010, pp. 1–5.
[25] R. Terra, M. T. Valente, S. Miranda, and V. Sales, "JMove: A novel heuristic and tool to detect move method refactoring opportunities," vol. 138, pp. 19–36, Apr. 2017.
[26] S. Charalampidou, A. Ampatzoglou, A. Chatzigeorgiou, A. Gkortzis, and P. Avgeriou, "Identifying extract method refactoring opportunities based on functional relevance," *IEEE Trans. Softw. Eng.*, vol. 43, no. 10, pp. 954–974, Oct. 2017.
[27] M. Fokaefs, N. Tsantalis, E. Stroulia, and A. Chatzigeorgiou, "Identification and application of extract class refactorings in object-oriented systems," *J. Syst. Softw.*, vol. 85, no. 10, pp. 2241–2260, 2012.
[28] S. Chandran and B. G. Varghese, "A survey on clustering techniques for identification of extract class opportunities," *Int. J. Res. Eng. Technol.*, vol. 2, no. 12, pp. 426–429, Dec. 2013.
[29] N. Tsantalis and A. Chatzigeorgiou, "Identification of move method refactoring opportunities," *IEEE Trans. Softw. Eng.*, vol. 35, no. 3, pp. 347–367, May 2009.
[30] D. E. Khelladi, R. Bendraou, and M.-P. Gervais, "AD-ROOM: A tool for automatic detection of refactorings in object-oriented models," in *Proc. IEEE/ACM 38th Int. Conf. Softw. Eng. Companion (ICSE-C)*, 2016, pp. 617–620.
[31] N. Tsantalis, V. Guana, E. Stroulia, and A. Hindle, "A multidimensional empirical study on refactoring activity," in *Proc. CASCON*, 2013, pp. 132–146.
[32] D. Silva, N. Tsantalis, and M. T. Valente, "Why we refactor? Confessions of GitHub contributors," in *Proc. 24th ACM SIGSOFT Int. Symp. Found. Softw. Eng. (FSE)*, New York, NY, USA, 2016, pp. 858–870.
[33] K. Prete, N. Rachatasumrit, N. Sudan, and M. Kim, "Template-based reconstruction of complex refactorings," in *Proc. IEEE Int. Conf. Softw. Maintenance*, Sep. 2010, pp. 1–10.

THIDA OO received the B.E. degree in information technology from Technological University, Monywa, Myanmar, in 2008, and the M.E. degree in information technology from Mandalay Technological University, Myanmar, in 2011. She is currently pursuing the Ph.D. degree with the School of Computer Science and Technology, Beijing Institute of Technology, China. She is currently a Staff Officer with the Department of Technical and Vocational Education and Training, Ministry of Education, Myanmar. Her research interests include software engineering and refactoring.



HUI LIU received the B.S. degree in control science from Shandong University in 2001, the M.S. degree in computer science from Shanghai University in 2004, and the Ph.D. degree in computer science from Peking University in 2008. He was a Visiting Research Fellow in Centre for Research on Evolution, Search and Testing, University College London, U.K. He is currently a Professor with the School of Computer Science and Technology, Beijing Institute of Technology, China. He served on the program committees and organizing committees for prestigious conferences, such as ICSME, RE, ICSR, and COMPSAC. He is particularly interested in software refactoring, AI-based software engineering, and software quality. He is also interested in developing practical tools to assist software engineers.

BRIDGET NYIRONGO received the M.Sc. degree in computer science and technology from the Beijing Institute of Technology, China, in 2016. She is currently a System Engineer with the Chancellor College ICT Centre, University of Malawi, specifically for software development. Her research interests are in analyzing the dynamic ranking of refactoring menu items.