Available online at www.ijpe-online.com

International Journal of Performability Engineering

vol. 16, no. 7, July 2020, pp. 1095-1104
DOI: 10.23940/ijpe.20.07.p12.10951104

A Prototype for Software Refactoring Recommendation System

Yuan Gao[a,*], Youchun Zhang[a], Wenpeng Lu[b], Jie Luo[c], and Daqing Hao[d]

[a]*China National-Local Joint Engineering Lab of Next Generation Internet Data Processing Technology*
University of Electronic Science and Technology of China, Chengdu, 610054, China
[b]*China School of Computer Science and Technology, Qilu University of Technology, Shandong Academy of Sciences, Jinan, 250353, China*
[c]*State Key Laboratory of Software Development Environment, School of Computer Science and Engineering, Beihang University, Beijing, 100191, China*
[d]*Luoyang Bearing Research Institute Co., Ltd., Luoyang, 471039, China*

Abstract

Software refactoring is used to reduce the costs and risks of software evolution. Automated software refactoring tools can reduce risks caused by manual refactoring, improve efficiency, and reduce difficulties of software refactoring. Researchers have made great efforts to research how to implement and improve automated software refactoring tools. However, results of automated refactoring tools often deviate from the intentions of the implementer. To this end, in this paper, we proposed and implemented a prototype tool for a software refactoring recommendation system based on previous research. The tool provides users with an optimized software refactoring scheme and users realize refactoring intentions by interacting with the tool. The tool has been evaluated to be effective, especially for users who are inexperienced and non- English speaking.

Keywords: software refactoring; recommendation; refactoring tool

(Submitted on April 26, 2020; Revised on May 24, 2020; Accepted on June 28, 2020)

1. Introduction

Software refactoring [1-3] is an effective technique to improve software quality. It is to improve readability, maintainability, and extendibility of software by adjusting its internal structure whereas the external characters of software are not changed. Since William F. Opdyke [1] proposed the conception of software refactoring for the first time in 1992, software refactoring has been a hot spot of research and has been well-recognized in industry and academia [4]. Researchers have carried out a large amount of research work on software refactoring [5-8].

However, software refactoring is a complicated activity of adjusting code. The process of manual refactoring is very complex, and it is very difficult and tedious. Software refactoring requires programmers to make various complex decisions, such as where to refactor, when to refactor, and how to refactor.

Although programmers know that they can benefit from software refactoring, the excessive cost makes people continue working on original code instead of software refactoring.

Automated software tools can relieve programmers of the burden of coding [9]. It reduces coding and debugging time, avoids human-induced errors, and makes programmers ignore cumbersome code-level details during software refactoring, thus improving efficiency and reducing costs.

Nevertheless, refactoring tools have not been widely used in the process of actual software development and maintenance. Most refactoring operations are still accomplished manually [10-11]. The reasons include two main points: First, some programmers are inexperienced in refactoring and unfamiliar with refactoring tools. They are more conservative when refactoring, and prefer to implement refactoring manually [12-14]. Secondly, programmers still need a strong decision-making ability to decide a refactoring solution by themselves in the process of using refactoring tools [15-16]. Even though, the final

* Corresponding author.
E-mail address: 306602175@qq.com

effect of software refactoring cannot be fully guaranteed.

Therefore, providing reasonable alternatives for programmers by automatic recommendation technology can reduce the difficulty and cost of refactoring. Furthermore, it can also provide reference and data support for programmers to make refactoring decisions, thereby improving the effect of refactoring [17-18].

In this paper, we proposed and implemented a prototype tool: SRRS(Software Refactoring Recommendation System) for a software refactoring recommendation system based on previous research [19-22] to assist programmers in making refactoring decisions, so as to reduce difficulty of refactoring and improve the effect of refactoring.

2. Motivation

The thought of the software refactoring recommendation system tool is to provide users with a refactoring recommendation scheme. Under the environment of the tool, users can decide the sequence and type of refactoring independently according to the refactoring scheme and their own refactoring intentions, so as to reduce difficulty and cost of refactoring. Therefore, in tool design, the main factors we concerned with are as follows.

2.1. Refactoring Opportunities Recommendation

Based on detection of programmer codes, the tool recommends refactoring actions which are taken to deal with bad smells hidden in codes.

2.2. Refactoring Priority Recommendation

Different orders of handling bad codes will not only affect efficiency and effectiveness of refactoring, but also lead to conflicts between refactorings. The tool can provide a refactoring scheme according to a degree of impact, cost of modification and other factors.

2.3. Test-Driven Refactoring Recommendation

Based on Test-Driven Development technology, the tool can recommend refactorings to users according to monitoring the changes of test code dynamically.

2.4. Keeping System Simple and Clear

As a basic application framework, further expansion and improvement should be considered in later stages. Therefore, it is necessary to keep simplicity and clarity of the system so that the system does not contain redundant codes and has simple and neat interfaces.

2.5. Achieving Reconfigurability of Internal Tools

Inside the tool, separate modules should be implemented for each method. This reduces complexity of the implementation and improves ease of use of the tool, which allows users to choose the appropriate tool as needed. In addition, in future expansion and improvement processes, new independent methods can be used in combination with existing methods to further improve the tool scalability.

2.6. Multiple Forms of Output

The output of any module in the tool should be easy to analyze. This allows the user to read and understand directly and helps to quickly obtain feedback for further work.

3. Implementation

The tool architecture is determined by software components, component attributes, and the relationship between components. It can be defined as a description which comprises a collection of components and interaction between components. The framework of the tool is designed based on eclipse.

As mentioned in motivation, architecture of the software refactoring recommendation system is depicted in Figure 1.

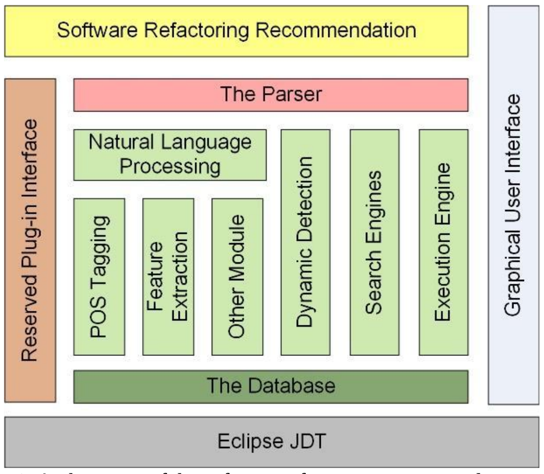


Figure 1. Architecture of the software refactoring recommendation system

3.1. Support Platform

The tool is designed based on Eclipse. Eclipse JDT (Java Development Tool) provides a set of application interfaces for accessing and controlling Java source code, and it is a set of extensions to the workbench. JDT accesses and controls Java code in two ways: Java Model and Abstract Syntax Tree. Currently, it mainly supports analysis of Java projects, setting build path, customizing compiler settings, supporting refactoring operations, building and running Java projects, and control of debugging Java projects and views.

Therefore, in the design, we have achieved access to the database through an interface provided by Eclipse JDT, a developed graphical user interface, and an integrated natural language processing tool. At the same time, the open plug-in interface of the Eclipse Open Platform reserved interface for future tool expansion and improvement.

3.2. The Database

During the refactoring recommendation process, whether accessing pre-processed data, comparing data between intermediate processes, or matching heuristic rules, many data needs to be accessed. It also generated a large number of intermediate data. The database can not only store a large amount of data information structurally and facilitate users to effectively retrieve and access, but it can also effectively maintain consistency, integrity and reduce data redundancy of data information. We manage these and data information through databases. The database provides an entity interface for storing, retrieving, and accessing data, as depicted in Figure 2.

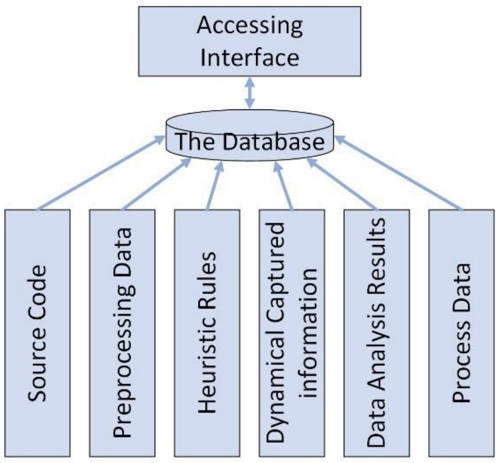


Figure 2. The use of the database

3.3. Natural Language Processing

In the tool, we integrated methods for dealing with bad smells such as inappropriate naming. For such bad smells, it is necessary to coordinate people's judgment with linking the context and making semantic analysis in order to make refactoring decisions. Normal code analysis and measurement methods have little effect on such bad smells. Therefore, natural language processing tools can be used to pre-process the code before it is analyzed by the parser, which lays a foundation for further analysis of the code by means of semantics analysis and parsing.

3.4. Dynamic Detection

Dynamic detection provides an online way to monitor code changes in real time and speculate about possible refactoring opportunities based on predefined heuristic rules. This allows users to get information about dynamic changes of code in real time, without need for additional tools or comparisons between different versions of code manually. However, the dynamic detection we are currently implementing can only detect changes in code modifications in stages. This module will be improved in future work.

3.5. Search Engines

Search engines played a major role in matching code and rules. They have a greater impact on accuracy of results. Similarity thresholds have been discussed experimentally. Experiments show that a single error has little effect on the system. However, if errors are superimposed, the accuracy of results will be greatly affected. Consequently, to improve the search engine algorithm is also an important direction for future work.

3.6. Execution Engine

The execution engine is responsible for scheduling different recommendation algorithms according to different software refactoring goals for users. Currently, we only implement three kinds of engines: recommending method names based on the source code depository and feature matching, recommending refactoring from test case changes, and resolution sequence for software refactoring based on severity priority. In future work, we will integrate new methods into the tool in conjunction with reserved interfaces and assign different tasks to the execution engine.

3.7. The Parser

The parser consists of two parts: Java static parser and XML parser. The structure of the Java static parser is depicted in Figure 3, which consists of three parts: Java parser, abstract syntax tree and structure/variable parser.

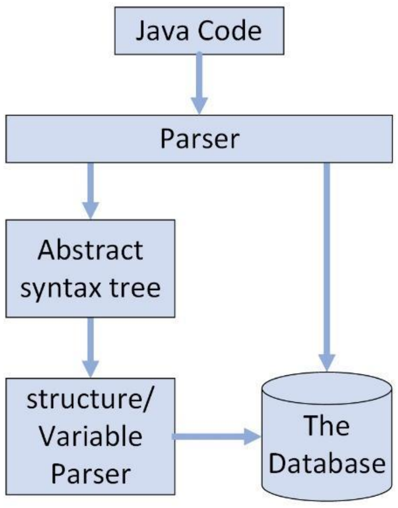


Figure 3. The structure of the Java Static Parser

The Java parser generates an abstract syntax tree by parsing the program code (AST node class diagram is depicted in Figure 4), while the structure/variable parser parses structure of program code. Finally, results of analysis of the two parts are stored in the database as XML.

During the process of system operation, we interpret information stored as XML in the database and convert it into the required abstract syntax tree. We designed an XML parser to accomplish this process.

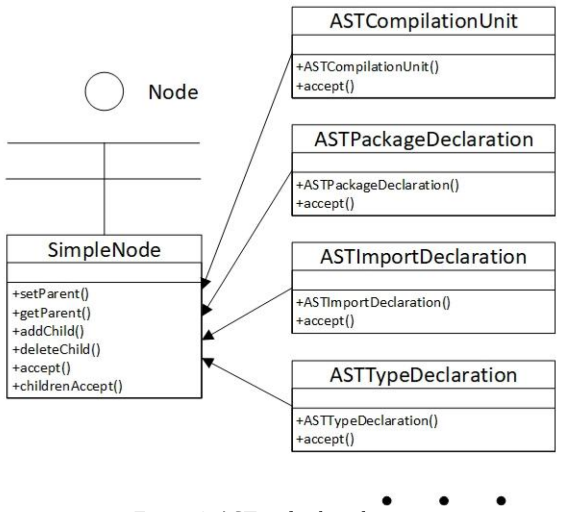


Figure 4. AST node class diagram

3.8. Refactoring Recommendation

The refactoring recommendation module is primarily used to determine in what form the generated results will be presented to the user. First, it needs to make judgments about from which method the analysis data is derived. Then, according to type of analysis, results are encapsulated into appropriate forms that are easy for users to understand and analyze. The user decides what to do next based on recommended results.

3.9. Graphical User Interface

In the graphical user interface module, we use an "event-driven" approach, which allows direct feedback to users when results are obtained from data analysis through an interface provided by eclipse JDT. The schematic diagram of classes involved in graphical user interface is shown in Figure 5. We divide the program into two parts: result capture code and event handling. The former is used to define behavior of components to capture results, and the latter is used to respond to user processing. In this way, different UI components are encapsulated in different classes, which conform to business logic of the program itself.

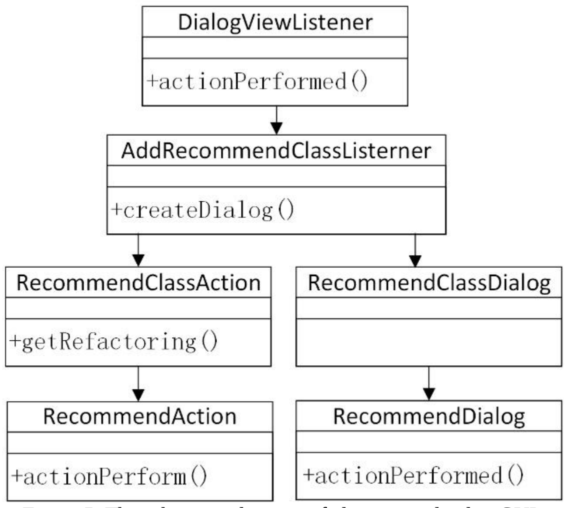
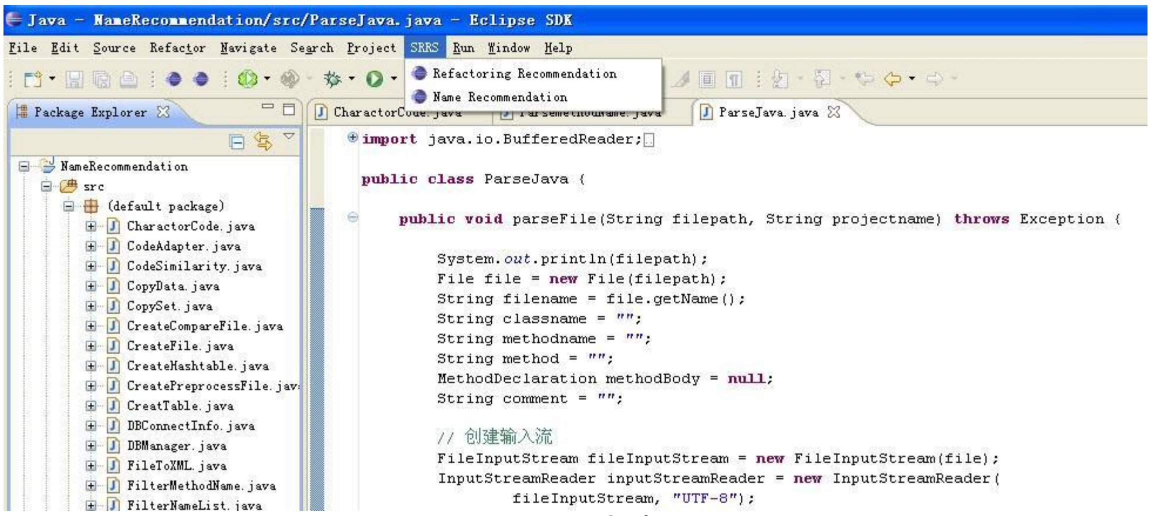


Figure 5. The schematic diagram of classes involved in GUI

As depicted in Figure 6, we take the refactoring recommendation and function name recommendation as examples. The tool item SRRS is added to the main menu bar of Eclipse. When the item is selected, the sub-menu item will be activated, which contains two menu items: Refactoring Recommendation and Name Recommendation, corresponding to refactoring recommendation and naming recommendation respectively.

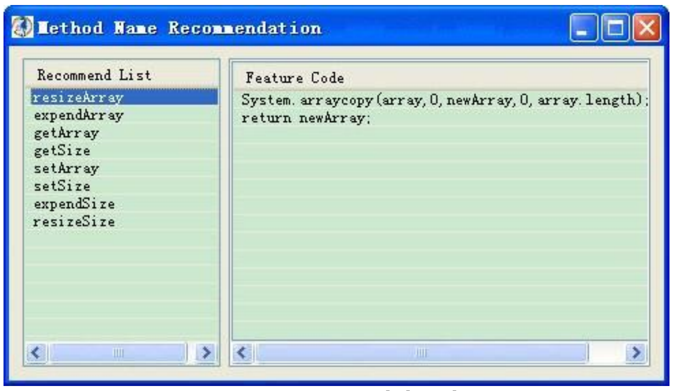
When a user uses the tool for refactoring recommendation, he can select the Refactoring Recommendation menu item.

Then, the system will run in the background and monitor the status of the currently edited code. When possible refactoring is detected, it will prompt the user to refactor. When performing the name recommendation refactoring, the user can in turn select the function code and use the Name Recommendation menu item to recommend names for the user. The system will retrieve the database based on the source code depository and feature matching. Recommended results will be listed for the user, as depicted in Figure 7.



```
import java.io.BufferedReader;

public class ParseJava {

    public void parseFile(String filepath, String projectname) throws Exception {

        System.out.println(filepath);
        File file = new File(filepath);
        String filename = file.getName();
        String classname = "";
        String methodname = "";
        String method = "";
        MethodDeclaration methodBody = null;
        String comment = "";

        // 创建输入流
        FileInputStream fileInputStream = new FileInputStream(file);
        InputStreamReader inputStreamReader = new InputStreamReader(
                fileInputStream, "UTF-8");
```

Figure 6. A sample of GUI



Recommend List
resizeArray
expendArray
getArray
getSize
setArray
setSize
expendSize
resizeSize

```
System.arraycopy(array,0,newArray,0,array.length);
return newArray;
```

Figure 7. Recommended results

3.10. Reserved Plug-in Interface

The tool is designed based on an open Eclipse platform and follows an open interface standard of Eclipse plug-ins. New plug-ins can be easily integrated into this tool, which is conducive to choosing an appropriate plug-in according to the actual needs and further improvement and expansion in future.

4. Experimental Evaluation

To evaluate the tool, two experiments are conducted.

First, we selected two medium-sized open source projects as validation objects: ***jEdit*** and ***text_Editor***. 295 functions in ***jEdit*** and 1135 functions in ***text_editor*** are verified. The library was constructed from 533,000 functions selected from 55 similar open source projects. We classify recommended results into five categories after comparing them with original function name: (1) identical with original; (2) identical with original keyword; (3) basically identical with original keyword; (4) different from original; (5) no recommend result. Results show that 85.29% of recommended results are available for 295 functions in ***jEdit*** and 80.67% of recommended results are available for 1135 functions in ***text_Editor***.

Second, we selected two test-driven development projects (***JRetrieval*** and ***JFramework***). In the actual development of the project, the effectiveness of the tool is verified by comparing the recommendation results with the actual refactoring intention. At the same time, we compared the results with Refiner's. The recommended results are presented in terms of accuracy and recall. Experimental results show that the accuracy rate reaches 87.5% for ***JRetrieval*** and 90.6% for ***JFramework***. Compared with Refiner, the accuracy is improved by about 15% and the recall rate is improved by 33.3-42.8%.

The experiment shows that the tool can effectively assist users in making refactoring decisions, thereby reducing the cost of refactoring.

5. Related Work

Bad smell [3] refers to the program code that affects software structure due to design defects or bad coding habits. It summarizes some common patterns of software problems and points out potential problems in code. Users can find and locate problems through bad smells and make clear where to implement refactoring and what kind of refactoring to implement. Therefore, some researches recommend software refactoring opportunities by detecting bad smell.

It is a common method to detect bad smell by detecting changes in the abstract syntax tree. Roberts [23] points out that the abstract syntax tree with type and reference information is an ideal form to represent a refactoring program; rich information contained in the abstract syntax tree is enough for software refactoring and can be created quickly. Koschke et al. [24] uses an abstract syntax suffix tree to detect cloned code.

Visualization Technology abstracts programs at different levels according to different requirements based on which detects bad smells. Simon [25] uses measure-based visualization technology to abstract cohesion of a program on a 2D graph to assist maintainers in identifying software refactoring. Emden and Moonen [26] propose a method for automatic detection and visualization of bad smells. Based on the method, they designed and implemented a prototype tool jCOSMO, which displays the code structure as a graph and associates bad smell with graph attributes. Lanza and Ducasse [27] proposed an evolutionary model that combines software metrics with visualization lightweight for the evolutionary pattern of object-oriented software. Bohnet [28] implements a prototype tool that dynamically substitutes layer-by-layer call relationships between code components through visual development methods. By analyzing the call relationship, we can judge its rationality and identify possible bad smells. Parnin et al. [29] proposed a visualization method to detect bad smell. They proposed a lightweight and logical catalog of bad smell and designed a visual studio plug-in, NOSEPRINTS, to find and display bad smell.

Slicing technology is always used to detect duplicate code. Komondoor and Horwitz [30] divide statements in a program into several equivalent classes according to their syntax structure. For elements in each pair of equivalent classes, isomorphic subgraphs are found to detect duplicate codes by performing backward and forward slicing analysis on the program dependency graph. Komondoor [31] proposes a method of duplicate code detection based on slicing technology. The method performs a series of transformations to the program through a process extraction algorithm, which is better for detecting some complex duplicate codes such as disorder sequence.

In order to improve usability of software refactoring tools and reduce difficulty of software refactoring, researchers speculate intention of software refactoring of programmers by detecting code changes and then recommend refactoring, thus assisting programmers in software reconfiguration. Ge and Murphy-Hill [32-33] proposed a method to infer users' refactoring intentions by detecting behavior of manual refactoring. The method consists of two parts: refactoring detection and code adjustment. Based on this approach, they implemented the software refactoring tool BeneFactor. Similarly, WitchDoctor [34] infers the user's refactoring intent by manually modifying code preliminarily, thus providing the user with a series of code conversion operations and completing relevant work of software refactoring. Raychev et al. [35] proposed a software refactoring method based on comprehensive examples. Based on the abstract syntax tree, this method detects changes in code and uses a search strategy based on the concept of local refactoring to find refactoring including the user's editing action, so as to automatically complete the process of software refactoring. Negara et al. [36-37] proposed an algorithm to infer the intent of software refactoring based on the abstract syntax tree. The algorithm distinguishes AST nodes before and after modification by setting unique IDs. Then, the intent of software refactoring is inferred by monitoring the change of attributes of matched AST nodes compared with predefined software refactoring features. Weibgerber et al [38] proposed a method to remind users that refactoring is being implemented. This method compares current code with previous versions, identifies refactoring candidates by the signature based method, and ranks and filters candidates by checking the similarity between codes to recommend the final results to the user. Yoshida et al. [39] proposed a method to assist users in refactoring cloned code by analyzing code adjustments online. The method detects cloned code incrementally as user editing. It captures the user's refactoring information, detects the same cloned code from existing code and prompts the user whether to refactor the cloned code. Jiau and Chen [40] proposed the concept of test-driven refactoring and proposed a method for inferring software refactoring from different versions of test code. According to certain matching principles, this method selects matching test cases from two versions of test cases, and then infers possible refactoring by comparing changes of test cases. The matching principle of selecting test cases will result in the omission of software refactoring information, and algorithm particles used to select test code matching nodes are too large, which leads to low accuracy and recall.

Software refactoring is used to reduce cost and risk of software evolution. Automated software refactoring recommendation tools can reduce risk caused by manual refactoring, improve efficiency of software refactoring, and reduce difficulty of software refactoring. Therefore, researchers discussed how to implement software refactoring recommendation tools and implemented corresponding software refactoring recommendation tools based on the proposed methods. Happel and maalej [41] discussed how to build a software refactoring recommendation system. They insist that the two main points should be focused: when to recommend and what to recommend. Their research has a certain guiding significance for the construction of software refactoring recommendation tools. Campbell and Miller [42] proposed a method for improving software refactoring tools. They propose a method of using source code refactoring feasibility tests to analyze code in background and recommend refactoring, and use a prompt system to visualize possible changes before users implement refactoring. Based on the method, they implemented a refactoring tool, Refactor! Pro [43], in Visual Studio development platform. Liu et al. [44] proposed a monitoring-based method to detect and prompt bad smells during development, thus driving developers to deal with them. In this method, a monitor runs in the background to monitor change of code. Once the current code change is considered to introduce bad smell, optimized bad smell detection algorithm is called, and the detected bad smell is recommended to developers to assist developers in implementing software refactoring. Based on this method, they implemented a relevant tool, InsRefactor. Bavota et al. [45] proposed a method based on a team collaborative maintenance model to identify and recommend software refactoring. This method excavates and identifies developers of the same team from historical data of the project. Code adjustment of the same team of developers is analyzed at different granularity according to different types of software refactoring. They implemented a corresponding tool, TBR. Silva et al. [46] proposed a method to identify and recommend extraction method refactoring. The method uses a hierarchical model to represent the structure of method blocks and a retrieval algorithm to select candidates for the refactoring extraction method. Dependency of the extraction part is then calculated by comparing variables, types, and package information of the extraction part and legacy part. Refactoring is recommended according to the degree of dependency. They integrated the method with the tool JMove [47]. Compared with JDeodorant [48], the tool can effectively improve the accuracy and recall rate. Mkaouer et al. [49] proposed a dynamic interactive method to recommend refactoring. The method uses a multi-objective optimization algorithm, NSGA-II, to calculate the optimal sequence of refactoring. Three strategies are used to recommend refactoring: refactoring times, location and user's feedback. They implement a tool, Dinar.

6. Conclusion

The process of software refactoring requires programmers to make continuous decisions to achieve the optimal effect of software refactoring. In order to reduce the difficulty and improve efficiency and effect of software refactoring, researchers put forward some recommended methods and implemented relevant tools to assist programmers in the software refactoring decisions-making. This reduces the manual workload of programmers in the process of software refactoring and provides programmers with various refactoring schemes, thus reducing the difficulty of software refactoring and improving the efficiency and effect of software refactoring. However, due to the large number of decision points involved in the process of software refactoring, existing recommended methods and tools cannot fully cover all decision points, or recommended results that are unsatisfactory.

To this end, based on previous research, we implement a software refactoring recommendation tool to assist programmers in refactoring decision-making. Experimental validation shows that the tool can effectively assist users in the refactoring decision, thus reducing refactoring cost.

References

1. W. F. Opdyke, “Refactoring Object-Oriented Frameworks,” Ph.D. Thesis, Urbana-Champaign, IL, USA, 1992
2. T. Mens and T. Tourw´e, “A Survey of Software Refactoring,” *IEEE Transactions on Software Engineering*, Vol. 30, pp. 126-139, February 2004
3. M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, “Refactoring: Improving the Design of Existing Code,” Addison-Wesley Professional, 2003
4. N. Hajrahimi and S. M .H. Dehaghani, “Which Factors Affect Software Projects Maintenance Cost More?” *Journal of Academy of Medical Sciences of Bosnia and Herzegovina*, Vol. 21, pp. 63-66, March 2013
5. Y. Kataoka, T. Imai, H. Andou, and T. Fukaya, “A Quantitative Evaluation of Maintainability Enhancement by Refactoring,” in *Proceedings of the International Conference on Software Maintenance*, pp. 576-585, Washington, USA, October 2002
6. M. Kim, D. Cai, and S. Kim, “An Empirical Investigation into the Role of Api-Level Refactorings During Software Evolution,” in *Proceedings of the 33rd International Conference on Software Engineering*, pp. 151-160, Waikiki, Honolulu, USA, May 2011
7. M. Kim, T. Zimmermann, and N. Nagappan, “A Field Study of Refactoring Challenges and Benefits,” in *Proceedings of the ACM*

SIGSOFT 20th International Symposium on the Foundations of Software Engineering, pp. 1-11, Cary North Carolina, November 2012

8. F. A. Fontana and S. Spinelli, “Impact of Refactoring on Quality Code Evaluation,” in *Proceedings of the 4th Workshop on Refactoring Tools*, pp. 37-40, Waikiki, Honolulu HI, USA, May 2011
9. E. Murphy-Hill and P. B. Andrew, “Refactoring Tools: Fitness for Purpose,” *IEEE Software*, Vol. 25, No. 5, pp. 38-44, September 2008
10. E. Murphy-Hill, C. Parnin, and A. P. Black, “How We Refactor, and How We Know It,” in *Proceedings of the 31st International Conference on Software Engineering, ICSE'09*, pp. 287-297, Washington, DC, USA, May 2009,.
11. E. Murphy-hill and A. P. Black, “Why Don't People Use Refactoring Tools,” in *Proceedings of the 1st Workshop on Refactoring Tools, ECOOP'07*, pp. 60-61, TU Berlin, July 2007
12. S. Diehl, P. WeiBgerber, and B. Biegel, “Making Programmers Aware of Refactorings,” *WRT 2007*, pp. 58-59, TU Berlin, July 2007
13. E. Murphy-Hill, R. Jiresal, and G. C. Murphy, “Improving Software Developers' Fluency by Recommending Development Environment Commands,” in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, FSE'12*, pp. 1-11, Cary, North, Carolina, November 2012
14. G. H. Pinto and F. Kamei, “What Programmers Say about Refactoring Tools?: An Empirical Investigation of Stack Overflow,” in *Proceedings of the 2013 ACM Workshop on Workshop on Refactoring Tools, WRT'1*3, pp. 33-36, Indianapolis, Indiana, USA, October 2013
15. X. Ge and E. Murphy-Hill, “Manual Refactoring Changes with Automated Refactoring Validation,” in *Proceedings of the 36th International Conference on Software Engineering, ICSE 2014*, pp. 1095-1105, Hyderabad, India, May 2014
16. E. Murphy-Hill and A. P. Black, “Breaking the Barriers to Successful Refactoring: Observations and Tools for Extract Method,” in *Proceedings of the 30th International Conference on Software Engineering, ICSE'08*, pp. 421-430, Leipzig, Germany, May 2008
17. A. P. Black and E. Murphy-hill, “Restructuring Software with Gestures,” in *Proceedings of IEEE Symposium on Visual Languages and Human-Centric Computing* (*VL*/*HCC*), pp. 165-172, 2011
18. E. Murphy-Hill, “Activating Refactorings Faster,” in *Proceedings of Companion to the 22Nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications Companion, OOPSLA'07*, pp. 925-926, Montreal, Quebec, Canada, October 2007
19. Y. Gao, H. Liu, X. Fan, and Z. Niu, “Analyzing Refactorings' Impact on Regression Test Cases,” in *Proceedings of the 2015 IEEE 39th Annual Computer Software and Applications Conference*, Vol. 2, pp. 222-231, July 2015
20. Y. Gao, H. Liu, X. Z. Fan, Z. D. Niu, and W. Z. Shao, “Resolution Sequence of Bad Smells,” *Journal of Software*, Vol. 23, pp. 1965-1977, August 2012
21. Y. Gao, H. Liu, X. Z. Fan, and Z. D. Niu, “Method Name Recommendation based on Source Code Depository and Feature Matching,” *Journal of Software*, Vol. 26, pp. 3062-3074, December 2015
22. Y. Gao, H. Liu, X. Z. Fan, and Z. D. Niu, “Inferring Refactoring Intention from Test Case Modification,” *Transactions of Beijing Institute of Technology*, Vol. 37, pp. 537-543, May 2017
23. D. B. Roberts, “Practical Analysis for Refactoring,” PhD Thesis, Champaign, IL, USA, 1999
24. R. Koschke, R. Falke, and P. Frenzel, “Clone Detection using Abstract Syntax Suffix Trees,” in *Proceedings of 13th Working Conference on Reverse Engineering*, pp. 253-262, Benevento, October 2006
25. F. Simon, F. Steinbrucker, and C. Lewerentz, “Metrics based Refactoring,” in *Proceedings of Europen Conference on Software Maintenance and Reengineering*, pp. 30-38, March 2001
26. E. Van Emden and L. Moonen, “Java Quality Assurance by Detecting Code Smells,” in *Proceedings of the Ninth Working Conference on Reverse Engineering, WCRE'02*, pp. 97-106, Richmond, VA, USA, December 2002
27. M. Lanza and S. Ducasse, “Understanding Software Evolution using a Combination of software Visualization and Software Metrics,” *Journal of L Objet*, Vol. 8, pp. 135-149, 2002
28. J. Bohnet and J. Dollner, “Analyzing Feature Implementation by Visual Exploration of Architecturally-Embedded Call-Graphs,” in *Proceedings of the 2006 International Workshop on Dynamic Systems Analysis, WODA'06*, pp. 41-48, Shanghai, China, May 2006
29. C. Parnin, C. Gorg, and O. Nnadi, “A Catalogue of Lightweight Visualizations to Support Code Smell Inspection,” in *Proceedings of the 4th ACM Symposium on Software Visualization, SoftVis'08*, pp. 77-86, Ammersee Germany, September 2008
30. R. Komondoor and S. Horwitz, “Using Slicing to Identify Duplication in Source Code,” in *Proceedings of the International Symposium on Static Analysis*, pp. 40-56, Berlin, July 2001
31. R. V. Komondoor, “Automated Duplicated Code Detection and Procedure Extraction,” PhD Thesis, 2003
32. Xi Ge and E. Murphy-Hill, “Benefactor: A Flexible Refactoring Tool for Eclipse,” in *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion, OOPSLA'11*, pp. 19-20, Portland, Oregon, USA, October 2011
33. Xi Ge, Q. L. DuBose, and E. Murphy-Hill, “Reconciling Manual and Automatic Refactoring,” in *Proceedings of the 2012 International Conference on Software Engineering, ICSE 2012*, pp. 211-221, Zurich Switzerland, June 2012
34. S. R. Foster, W. G. Griswold, and S. Lerner, “Witchdoctor: Ide Support for Realtime Auto-Completion of Refactorings,” in *Proceedings of the 2012 International Conference on Software Engineering, ICSE 2012*, pp. 222-232, Piscataway, NJ, USA, 2012
35. M. Sridharan, M. Vechev, V. Raychev, and M. Schafer, “Refactoring with Synthesis,” *ACM Special Interest Group on Programming Languages*, Vol. 48, No. 10, pp. 339-354, October 2013
36. N. Chen, R. E. Johnson, D. Dig, S. Negara, and M. Vakilian, “Is It Dangerous to Use Version Control Histories to Study Source Code Evolution?” in *Proceedings of the 26th European Conference on Object-Oriented Programming*, pp. 79-103, June 2012
37. S. Negara, N. Chen, M. Vakilian, R. E. Johnson, and D. Dig, “A Comparative Study of Manual and Automated Refactorings,” in

Proceedings of the 27th European Conference on Object-Oriented Programming, ECOOP'13, pp. 552-576, Berlin, Heidelberg, 2013

38. S. Diehl, P. WeiBgerber, and B. Biegel, "Making Programmers Aware of Refactorings," in *Proceedings of the 1st Workshop on Refactoring Tools*, *ECOOP'07*, pp. 58-59, TU Berlin, July 2007
39. N. Yoshida, E. Choi, and K. Inoue, "Active Support for Clone Refactoring: A Perspective," in *Proceedings of the 2013 ACM Workshop on Workshop on Refactoring Tools*, *WRT'13*, pp. 13-16, Indianapolis, Indiana USA, October 2013
40. H. C. Jiau and J. C. Chen, "Test Code Differencing for Test-Driven Refactoring Automation," *SIGSOFT Softw. Eng. Notes*, Vol. 34, No. 1, pp. 1-10, January 2009
41. H. Happel and W. Maalej, "Potentials and Challenges of Recommendation Systems for Software Development," in *Proceedings of the 2008 International Workshop on Recommendation Systems for Software Engineering*, *RSSE'08*, pp. 11-15, Atlanta, Georgia, November 2008
42. D. Campbell and M. Miller, "Designing Refactoring Tools for Developers," in *Proceedings of the 2nd Workshop on Refactoring Tools*, *WRT'08*, pp. 1-2, Nashville, Tennessee, October 2008
43. Refactor! pro, (http://www.devexpress.com/products/net/refactor)
44. H. Liu, X. Guo, and W. Z. Shao, "Monitor-based Instant Software Refactoring," *IEEE Transactions on Software Engineering*, Vol. 39, pp. 1112-1126, August 2013
45. G. Bavota, S. Panichella, N. Tsantalis, M. Di Penta, R. Oliveto, and G. Canfora, "Recommending Refactorings based on Team Co-Maintenance Patterns," in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, *ASE'14*, pp. 337-342, Vasteras, Sweden, September 2014
46. D. Silva, R. Terra, and M. T. Valente, "Recommending Automated Extract Method Refactorings," in *Proceedings of the 22nd International Conference on Program Comprehension*, *ICPC 2014*, pp. 146-156, Hyderabad, India, June 2014
47. R. Terra, L. F. Miranda, M. T. Valente, and V. Sales, "Recommending Move Method Refactorings using Dependency Sets," in *Proceedings of 2013 20th Working Conference on Reverse Engineering* (*WCRE*), pp. 232-241, Koblenz, Germany, October 2013
48. A. Chatzigeorgiou and N. Tsantalis, "Identification of Extract Method Refactoring Opportunities for the Decomposition of Methods," *Journal of Systems and Software*, Vol. 84, pp. 1757-1782, October 2011
49. M. Mkaouer, M. Kessentini, S. Bechikh, K. Deb, and M. O. Cinneide, "Recommendation System for Software Refactoring using Innovization and Interactive Dynamic Optimization," in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, *ASE'14*, pp. 331-336, Vasteras, Sweden, September 2014