2012 IEEE 36th International Conference on Computer Software and Applications

An Initial Study on Refactoring Tactics

Hui Liu*†, Yuan Gao* and Zhendong Niu*
*School of Computer Science and Technology, Beijing Institute of Technology, Beijing 100081, China
†Key Laboratory of High Confidence Software Technologies, Ministry of Education, Beijing 100081, China
Email:{liuhui08,yuangao,zniu}@bit.edu.cn

***Abstract*—Software refactoring might be done in two different tactics. The first one is XP-style small-step refactoring, also called floss refactoring. The other tactic, called root canal refactoring, is to set aside an extended period specially for refactoring. Floss refactoring, as one of the corner stones of XP, is well acknowledged. In contrast, root canal refactoring is doubted, especially by XP advocators. Despite the doubts, however, no large scale empirical study on refactoring tactics has been reported. In contrast to the doubts, cases of root canal refactoring have been reported from industry, e.g., Microsoft. Researchers from academe have also proposed various approaches to facilitating root canal refactoring. To this end, this paper would investigate the following questions. (1)How often are the two different tactics employed, respectively? (2) Is there any correlation between refactoring tactics and categories of refactorings? In other words, are some kinds of refactorings more likely than others to be done as floss refactorings or root canal refactorings? To answer these questions, we analyze refactoring histories collected by Eclipse Usage Data Collector (UDC). The data are collected from 753,367 engineers worldwide. Analysis results suggest that about 11.5 percent of refactorings collected by UDC are root canal refactorings, whereas others (88.5 percent) are floss refactorings. We also find that some kinds of refactorings, e.g., *Introduce Parameter*, are more likely than others to be performed as root canal refactorings.**

***Keywords*-Software Refactoring; Tactic; XP; Refactoring History; Empirical Study**

I. Introduction

A. Software Refactoring Tactics

Software refactoring is to restructure the internal structure of object-oriented software applications to improve their quality, especially their maintainability, extensibility and reusability [1], [2]. At the same time, software refactoring preserves the external behaviors of software applications. Software refactoring is widely used to delay the degradation effects of software aging and facilitate software maintenance. It is possible that software applications are repeatedly modified according to evolving requirements. As a result, the source code shifts from its original design structure, and it becomes complex, difficult to read or debug, and even harder to extend. To deal with this issue, software refactoring is employed to improve software applications' readability and extensibility by cleaning up bad smells in source code. With the popularity of XP, software refactoring is also employed within routine development phases besides software maintenance. To date, refactoring is one of the corner stones of XP [3].

Software refactoring might be carried in two different tactics [4]. The first one is to set aside an extended period specially for refactoring (called root canal refactoring [5]). As a result, the refactorings are interspersed with few other activities, and thus they look more dense at that period. The other tactic XP-style small-step refactoring. Murphy-Hill and Black [5] call it floss refactoring. To implement a new function, XP encourages engineers to quickly complete an initial implementation that can pass pre-defined test cases. And then XP encourages engineers to improve the implementation by refactoring. Because XP advocates short develop iterations, floss refactorings are usually heavily interspersed with other routine development activities, e.g., adding new functionalities or fixing bugs. As a result, floss refactorings look sparse in the whole lifetime of software development.

Compared to floss refactoring, root canal refactoring has the following two features:

1) First, software engineers have to look for refactoring opportunities throughout applications they are restructuring. It is not easy because modern software applications are large and complex. To this end, researchers have proposed many detection algorithms and tools to facilitate the task [1], [6], [7], [8], [9], [10], [11], [12], [13], [14]. In contrast, Floss refactoring in XP usually focuses on a single method or class the software engineer is currently working on, and thus these detection tools are rarely used with Floss refactoring tactic.
2) Second, numerous refactoring opportunities might be found in a single application, and the resolution of these opportunities had better be scheduled to achieve greatest quality improvement [15], [16], [17], [18], [19], [20]. In contrast, only a small number of refactoring opportunities could be found at a time with Floss refactoring tactic. Consequently, schedule approaches for software refactoring usually set root canal refactoring as their target scenario.

B. Doubts about root canal refactoring

Floss refactoring is advocated by XP advocators, and it has become one of the corner stones of XP. However, root canal refactoring is still in doubt, especially by XP

0730-3157/12 $26.00 © 2012 IEEE
DOI 10.1109/COMPSAC.2012.31

advocators. Fowler declares [21] that he is opposed to setting aside specific time for refactoring in almost all cases. [22], an Agile consultant, also doubts root canal refactoring.

However, cases of root canal refactoring from industry are reported. For example, Microsoft usually reserves about 20% of development efforts on thorough refactoring, which would begins at the release of a software system, and ends at the beginning of development of the next iteration (development of the next version) [23]. Consequently, it is possible that a large number of refactorings would be carried out in batch model by Microsoft. Our experiences in Newegg[1], a well known international e-business company, also suggest that some might practice batch refactoring frequently. Software development teams of the company develop and maintain an e-business system. After a new version of the system is released, the development of the next version usually does not begin immediately. Instead, an extended period is allocated to evaluate the new release and analyze new requirements. During the period, some software engineers are assigned to fixing bugs, but more are assigned to thoroughly refactoring the system. Software refactoring improves the extensibility and maintainability of the application. As a result, once new features are proposed and approved, these features can be easily and quickly implemented based on the current version. It is valuable because fast delivery is critical for highly competitive industries, such as e-business. Besides the cases of Microsoft and Newegg mentioned above, other similar cases have also been reported [24], [25].

Moreover, the prosperity in research on root canal refactoring also suggest that root canal refactoring might be popular. As discussed in Section I-A, bad smell detection and refactoring schedule are mainly used with root canal refactoring. Researchers have done lot of work in these fields [1], [6], [7], [8], [9], [11], [12], [13], [14], [16], [15], [17], [18], [19], [20], suggesting they have found quite a few users or potential users of root canal refactoring.

Despite the controversy, however, to the best of our knowledge there is no large-scale empirical study on refactoring tactics. Murphy-Hill and Black [5] declare that floss refactoring is more popular than root canal refactoring, and empirically find that only 2 percent of refactorings root canal refactoring [4]. However, the finding is based on only 40 Eclipse CVS commits which are manually analyzed by the authors [4]. To the best of our knowledge, these [5], [4] are the only two papers investigating refactoring tactics. These papers are significant and interesting. However, to make conclusions more convincing, a larger scale empirical study is indispensable.

To this end, we conduct an empirical study to answer the following questions:

(1) How often are the two refactoring tactics employed, respectively?

(2) Are some kinds of refactorings more likely to be done as root canal refactoring or floss refactoring?

In the empirical study, we investigate refactoring histories captured in real industry. Results suggest that about 11.5 percent of software refactorings are root canal refactoring whereas others (88.5 percent) are floss refactoring.

The rest of the paper is structured as follows. Section II presents the analysis on refactoring histories captured by Eclipse UDC. Section IV presents a short overview of related work, and Section V makes a conclusion.

II. Experimental Setup

In this section, we would introduce refactoring histories to be analyzed, and discuss how these data should be analyzed.

A. Data Analyzed

Data to be analyzed are provided by Eclipse Usage Data Collector (UDC)[2]. UDC records how individuals use Eclipse platform, including the following information: (1)loaded bundles;(2)invoked commands;(3)activated actions embedded in Eclipse menus or toolbars; (4)state change of perspectives;(5)state change of views;(6)state change of editors.

All refactoring commands integrated with Eclipse are monitored by UDC. Consequently, it is possible to recover refactoring histories from the data provided by UDC.

Each record captured by UDC is composed of seven fields: *UserId*, *What*, *Kind*, *BundleId*, *BundleVersion*, *Description*, and *Timestamp*. The following is a sample record: "*1129908,activated, perspective, org.eclipse.jdt.ui, ,org.eclipse.jdt.ui.JavaPerspective, 1255058277952*"

Meaning of each field is explained as follows:

- **UserId** identifies the workstation and workspace where this activity is performed. It is used to track all activities performed by the same user. For the sample case, UserId is *1129908*.
- **What** indicates what happens. For the sample record, a perspective was *activated*.
- **Kind** indicates what kind of bundle is involved. For the sample record, the bundle is a *perspective*.
- **BundleId** is the ID of the involved bundle (plug-in). For the sample record, the perspective that caused the event comes from *org.eclipse.jdt.ui* bundle.
- **BundleVersion** version of the bundle. For the sample record, nothing is provided.
- **Description** provides some information about the event. In the sample record, the source perspective's ID is provided (*org.eclipse.jdt.ui.JavaPerspective*). If a refactoring command is activated, the command ID would be presented in this field. Consequently, it is an important field for refactoring identification.
- **Timestamp** indicates when the event occurs. The number is millisecond timestamp from the user's

[1]http://www.newegg.com

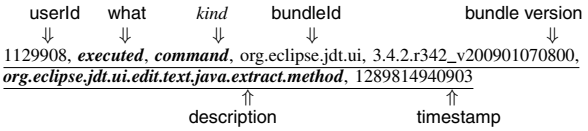
[2]http://www.eclipse.org/org/usagedata

workstation. For the sample case, the time stamp '1255058277952' suggests that the even happened at 2009-10-09 11:17:57.952.

Records captured between Jan., 2010 and May, 2010 are publicly available online[3] for academic research, and our investigation is done on these records. At the period, Eclipse UDC captured more than 615,788,499 activities performed by more than 753,367 users worldwide (the actual number of users might be a bit smaller than that of UserIDs because a UserID captured by UDC represents a unique workstation and workspace, and some users might work on multiple machines and workspace).

B. Identification of Refactorings

Once a refactoring command embedded in Eclipse is activated (suggesting a refactoring is carried out by the user), a record would be sent to UDC. The record would contain the command ID, constant strings 'executed', and 'command' in its fields *description*, *what*, and *kind*, respectively. For example, if refactoring command *extract method* is executed, a record similar to the following one would be captured:



Collecting all IDs of refactoring commands embedded in Eclipse[4], we can distinguish refactoring activities from others by comparing the following three fields: *description*,*what*, and *kind*.

It should be noted that if refactorings are not carried out by refactoring tools embedded in Eclipse, e.g., carried out manually, these refactoring activities could not be identified.

C. Identification of Root Canal Refactoring

Root canal refactorings are distinguished from floss refactorings in the following two steps.

1) First, a sequence of activities performed by the same user is segmented into small subsequences. Between successive subsequences, there must be an extended period (longer than half an hour) without any activity. The user is expected to leave Eclipse at the period for a break or something else. As a result of the segmentation, each subsequence represents a working segment, within which the user is actively working on Eclipse. With this approach, we identify 7,843,206 subsequences.
2) Second, if a number of successive subsequences of activities performed by the same user form an extended working period i.e.,longer than a half working day (4 hours), and within each of the subsequence the ratio of refactoring activities to all activities is higher than a predefined threshold (dense refactorings), we assume that the extended period is specifically set aside for refactoring, i.e., the user is carrying out root canal refactoring at the period.

It should be noted that even at the period specially set aside for software refactoring, software engineers might also do some activities that would be captured by UDC as non-refactoring activities. As a result, we cannot expect all activities performed at the period are refactoring activities. Instead, only a great ratio of refactoring activities (to all activities captured) is expected. The reasons are explained as follows.

- First, while carrying out refactorings, software engineers might explore the views of Eclipse, e.g., *Project View*, *Package View* or *Outline View*, to look for refactoring opportunities or to find the right places where refactorings are needed. These activities would be captured by UDC as non-refactoring activities.
- Second, while carrying out complex refactorings, software engineers might perform complex debugging and edits, e.g., *delete*, *copy*, *save*, and *run*. These activities would be captured by UDC as non-refactoring activities, too.
- Third, once refactorings are completed, test cases should be executed to make sure refactorings have been done correctly. Software testing would be captured by UDC as non-refactoring activities.
- Finally, refactorings carried out manually would be captured by UDC as non-refactoring activities.

D. Threshold Estimation

To estimate the minimal refactoring density (the ratio of refactoring activities to all activities) when an extended period is specifically set aside for root canal refactoring, we conduct the following case study.

Three software engineers with more than 3 years of experience with software refactoring are requested to set aside two days for root canal refactoring on projects they are currently working on. The first two engineers are developing a testing framework for railway systems, and they have work in this field for more than 5 years. The third engineer is developing an online education system, and he has work for the project for more than 2 years.

All refactorings are performed on Eclipse, and UDC is enabled to capture all activities on Eclipse. UDC stores the history of activities in local disks (default directory is \workspace\.metadata\.plugins \org.eclipse.epp.usagedata.recording) before it is uploaded to Eclipse server. Consequently, we do not have to request for these data from UDC managers.

Once refactorings are completed, we manually check the revision made to the projects to make sure that no functional change has been done, i.e., the whole period has been completely set aside for refactoring. After that, we

[3]http://archive.eclipse.org/technology/phoenix/usagedata/

[4]These IDs are provided by Wayne, the manager of UDC data

Table I
STATISTICS OF ANALYZED DATA

Items	Value
Num of Involved Users	753,367
Num of Activities	615,788,499
Num of Refactorings	629,178
Ratio of Refactorings to All Activities	0.1%
Num of Floss Refactorings	557,135
Ratio of Floss Refactorings to All Refactorings	88.5%
Num of Root Canal Refactorings	72,043
Ratio of Root Canal Refactorings to All Refactorings	11.5%

calculate the ratio of refactoring activities to all activities captured by UDC. The ratios are 0.81%, 1.02%, and 1.22% respectively. The ratios are small, and we manually explored records captured by UDC. We found that numerous activities assisting refactoring commands are captured by UDC as non-refactoring activities. The reasons have been discussed in Section II-C. We adopt the mean value 1% as the minimal refactoring density.

III. RESULTS AND FINDINGS

With the identification approaches proposed in Section II-B and Section II-C, and the threshold estimated in Section II-D, we try to analyze refactoring history collected by UDC in this section.

Statistics of the analyzed data are presented in Table I. The number of users monitored by UDC is 753,367. The number of activities captured by UDC is 615,788,499. 692,178 of the activities are refactorings, and among these refactorings 72,043 are root canal refactorings.

Detailed analysis of the statistics is presented and discussed as follows.

A. Relative Popularity of the Tactics

72,043 out of 629,178 refactoring activities are root canal refactorings. The ratio of root canal refactorings to all refactorings is $72,043/629,178 = 11.5\%$. In other words, more than 88% of refactorings are floss refactorings. The result is consistent with the declaration that floss refactoring is more common than root canal refactoring [4].

However, the overall ratio of root canal refactorings (11.5%) is much higher than that (2%) reported in [4]. One of the possible reasons for this huge difference is that the two empirical investigations involve different engineers who might adopt different refactoring tactics. Another possible reason is that our analysis involved much more software engineers and refactoring activities. In their investigation reported in [4], only 40 commits of Eclipse are analyzed to discover their refactoring tactics. In contrast, our investigation involve millions of activities performed by thousands of engineers worldwide working on different projects.

B. Variation among Different Kinds of Refactorings

For different kinds of refactorings, engineers might take different tactics. Consequently, ratio of root canal refactorings might vary among different kinds of refactorings. For those directly accessible from Eclipse menu *Refactor*, the ratios are presented in Fig. 1. From this figures, we observe that the ratio varies from 10.33% to 45.26% (no instance of *pull down* has been captured, and thus its ratio of batch refactorings is unavailable). Some kinds of refactorings, e.g., *introduce parameter* and *inline*, are more likely than others to be carried out as root canal refactoring. In contrast, some kinds of refactorings, e.g., *extract class* and *use super type*, are more likely than others to be done as floss refactoring.

The finding is interesting and valuable to researchers from academe who are proposing approaches and tools to facilitate root canal refactoring. With this finding, they can focus on those refactorings that are most likely to be done as root canal refactoring. The same is true for those who are investigating how to facilitate floss refactoring.

The reasons why some of them are more likely than others to be done as root canal refactoring have not been investigated yet. It would be interesting to dig into this issue in the near future. The investigation might uncover why and when a refactoring tactic is employed.

IV. RELATED WORK

A series of significant papers in this field have been published by Murphy-Hill and his colleagues. Murphy-Hill and Black [5] investigate how refactoring tools are used and what kind of refactoring tools would be more useful. They also define and compare the two refactoring tactics. They also declare that floss refactoring is more popular than root-canal refactoring. However, no validation has been provided in that paper. In this paper, we validate this statement with a large scale empirical study.

Murphy-Hill et al. [4] validate some assumptions on how refactorings are performed. One of the interesting findings they reported is that floss refactoring is much more frequent than root canal refactoring. Though multiple data sources (refactoring activities captured by Maylar [26] from 41 volunteer programmers, activities captured by Eclipse UDC, refactoring histories from 4 developers, and Eclipse CVS) are analyzed in their paper, however, this conclusion is validated on only 40 Eclipse CVS commits (20 labeled and 20 unlabeled) and other data sources are analyzed for other purposes. To make the validation more convincing, we validate it on 615,788,499 activities captured by UDC from 753,367 users worldwide. Moreover, we also make a further investigation on root canal refactoring, and some interesting

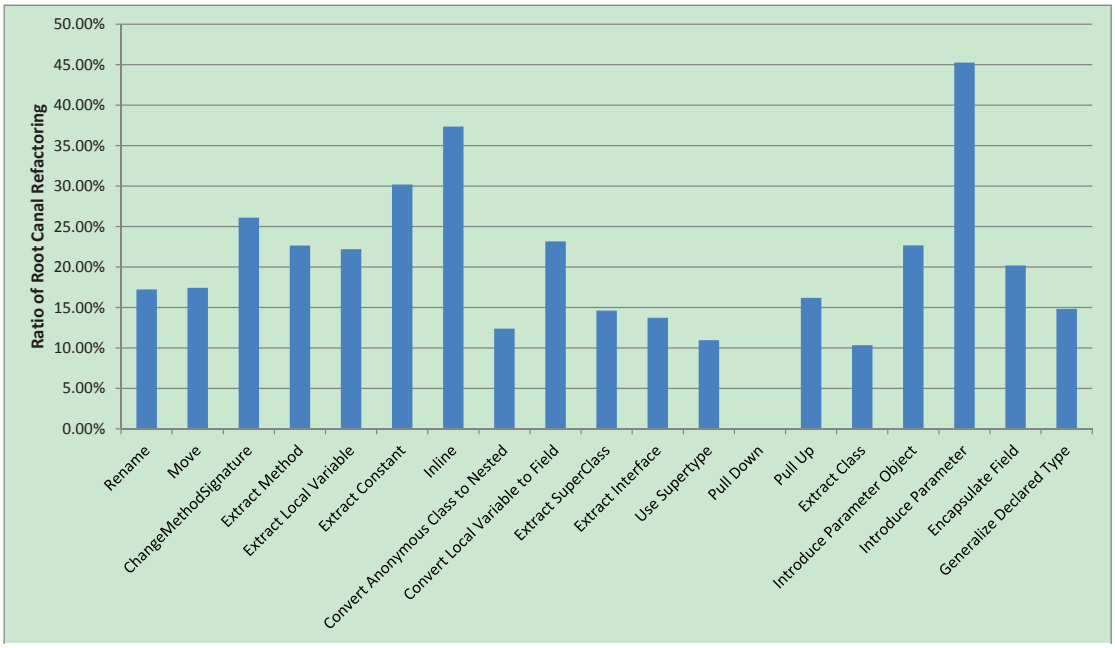


Figure 1. Ratio of Batch Refactorings Directly Accessible from Eclipse Menus

findings are reported. Details of these findings are presented in Section III.

Murphy-Hill et al. [27] list and discuss four methods for refactoring identification: Mining the commit log, analyzing code histories, observing programmers, and mining refactoring tools. According to their comparison, each of the methods has its own strength and weakness. Inspired by this work, in this paper we identify refactorings by mining UDC data (mining logs of refactoring tools).

Murphy et al. [26] investigate how java engineers use Eclipse via a plug-in Mylar Monitor collecting activities from 41 programmers. They also investigate how refactorings are done and which refactorings are performed more frequently. Our work is similar to theirs in that refactoring activities are captured by monitors integrated in users' IDE, UDC in our case and Mylar Monitor in their case. However, our work differs from theirs in that we investigate refactoring tactics whereas they investigate how users use Eclipse IDE and how frequently each refactoring command is activated. Finally, the scale of our investigation is much larger, involving nearly 753,367 software engineers.

Xing and Stroulia [28] report an Eclipse case study investigating refactoring practice and its tool support. They identify refactorings via UMLDiff [29] which implements a design-level structural differencing algorithm. They found that about 16% of all changes can be expressed in terms of refactorings, which is much higher than that (0.1%) found in this paper. One of the possible reasons is that UMLDiff [29] detects class-level changes only (which can be found in UML class diagrams), and changes within methods are not detected. Moreover, the focus of this paper is to investigate refactoring tactics which is not the topic of their work [28].

V. Conclusion and Future Work

To make software refactoring more effective, we should know more about how refactorings are performed. In this paper, we investigate how software engineers practice refactorings with a focus on refactoring tactics. We analyze activities captured by Eclipse UDC. Results suggest that about 11.5% of refactorings captured by UDC were root canal refactorings, and some kinds of refactorings are much more likely than others to be done as root canal refactorings.

Acknowledgments

The work is funded by the National Natural Science Foundation of China (No.61003065), Specialized Research Fund for the Doctoral Program of Higher Education (No.20101101120027), and Excellent Young Scholars Research Fund of Beijing Institute of Technology (No.2010Y0711).

References

[1] T. Mens and T. Touwe, "A survey of software refactoring," *IEEE Transactions on Software Engineering*, vol. 30, no. 2, pp. 126–139, 2004.

[2] W. F. Opdyke, "Refactoring object-oriented frameworks," Ph.D. dissertation, University of Illinois at Urbana-Champaign, 1992.

[3] K. Beck, *Extreme Programming Explained: Embrace Change*. Addison Wesley, 2000.

[4] 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*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 287–297.

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

[6] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: A multilinguistic token based code clone detection system for large scale source code," *IEEE Transactions on Software Engineering*, vol. 28, no. 6, pp. 654–670, 2002.

[7] Y. Kataoka, M. Ernst, W. Griswold, and D. Notkin, "Automated support for program refactoring using invariants," in *Proceedings of the 17th IEEE International Conference on Software Maintenance (ICSM'01)*, 2001, pp. 736 – 743.

[8] N. Moha, Y.-G. Gueheneuc, and P. Leduc, "Automatic generation of detection algorithms for design defects," in *Automated Software Engineering, 2006. ASE '06. 21st IEEE/ACM International Conference on*, Sept. 2006, pp. 297–300.

[9] N. Moha, Y.-G. Gueheneuc, L. Duchien, and A.-F. Le Meur, "Decor: A method for the specification and detection of code and design smells," *Software Engineering, IEEE Transactions on*, vol. 36, no. 1, pp. 20 –36, jan.-feb. 2010.

[10] N. Tsantalis and A. Chatzigeorgiou, "Identification of move method refactoring opportunities," *IEEE Transactions on Software Engineering*, vol. 99, no. RapidPosts, pp. 347–367, 2009.

[11] F. Tip, A. Kiezun, and D. Baeumer, "Refactoring for generalization using type constraints," in *Proceedings of the Eighteenth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'03)*, Anaheim, CA, October 2003, pp. 13–26.

[12] T. Tourwe and T. Mens, "Identifying refactoring opportunities using logic meta programming," in *Proceedings of the Seventh European Conference On Software Maintenance And Reengineering (CSMR'03)*, 2003, pp. 91–100.

[13] B. Van Rompaey, B. Du Bois, S. Demeyer, and M. Rieger, "On the detection of test smells: A metrics-based approach for general fixture and eager test," *Software Engineering, IEEE Transactions on*, vol. 33, no. 12, pp. 800–817, Dec. 2007.

[14] M. Munro, "Product metrics for automatic identification of "bad smell" design problems in java source-code," in *Software Metrics, 2005. 11th IEEE International Symposium*, Sept. 2005, pp. 15–15.

[15] H. Liu, Z. Ma, W. Shao, and Z. Niu, "Schedule of bad smell detection and resolution: A new way to save effort," *IEEE Transactions on Software Engineering*, vol. 38, no. 1, pp. 220–235, JANUARY/FEBRUARY 2012.

[16] S. Bouktif, G. Antoniol, E. Merlo, and M. Neteler, "A novel approach to optimize clone refactoring activity," in *Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation (GECCO'06)*, M. Cattolico, Ed., ACM SIGEVO (formerly ISGEC). Seattle USA: ACM Press, July 2006, pp. 1885–1892.

[17] H. Liu, G. Li, Z. Ma, and W. Shao, "Scheduling of conflicting refactorings to promote quality improvement." in *22nd IEEE/ACM International Conference on Automated Software Engineering (ASE'07)*, 2007, pp. 489 – 492.

[18] Hui Liu, Ge Li, Zhiyi Ma, and Weizhong Shao, "Conflict aware scheduling of software refactorings," *IET Software*, vol. 2, no. 5, pp. 446–460, Oct. 2008.

[19] H. Liu, L. Yang, Z. Niu, Z. Ma, and W. Shao, "Facilitating software refactoring with appropriate resolution order of bad smells," in *Proceedings of 7th joint meeting of the European Software Engineering Conference (ESEC) and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'09)*, 2009, pp. 265–268.

[20] W. C. Wake, *Refactoring Workbook*. Addison Wesley, August 2003.

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

[22] J. Shore, "Design debt," *Software Profitability Newsletter*, 01 Feb. 2004, http://jamesshore.com/Articles.

[23] M. A. Cusumano and R. W. Selby, *Microsoft Secrets*. Free Press, 1995.

[24] E. Stroulia and R. Kapoor, "Metrics of refactoring-based development: An experience report." in *Proc. of the 7th International Conference on Object-Oriented Information System*. Springer Verlag, 2001, p. 113C122.

[25] R. Weber, T. Helfenberger, and R. K. Keller, "Fit for change: Steps towards effective software maintenance," in *Proceedings of the 21st IEEE International Conference on Software Maintenance - Industrial and Tool volume*, 2005, pp. 26–33, recipient of Best Industrial Paper Award.

[26] G. C. Murphy, M. Kersten, and L. Findlater, "How are java software developers using the eclipse ide?" *IEEE Softw.*, vol. 23, pp. 76–83, July 2006. [Online]. Available: http://portal.acm.org/citation.cfm?id=1159169.1159396

[27] E. Murphy-Hill, A. P. Black, D. Dig, and C. Parnin, "Gathering refactoring data: A comparison of four methods," in *Proceedings of the 2nd Workshop on Refactoring Tools*, ser. WRT '08. New York, NY, USA: ACM, 2008, pp. 7:1–7:5. [Online]. Available: http://doi.acm.org/10.1145/1636642.1636649

[28] Z. Xing and E. Stroulia, "Refactoring practice: How it is and how it should be supported - an eclipse case study," *Software Maintenance, IEEE International Conference on*, pp. 458–468, 2006.

[29] Z. Xing and Eleni Stroulia, "Umldiff: an algorithm for object-oriented design differencing," in *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, 2005, pp. 54–65, xing, Zhenchang and Stroulia, Eleni.