Published in IET Software
Received on 7th August 2012
Revised on 21st June 2013
Accepted on 24th July 2013
doi: 10.1049/iet-sen.2012.0121

ISSN 1751-8806

Case study on software refactoring tactics

Hui Liu[1,2], *Yang Liu*[1], *Guo Xue*[1,3], *Yuan Gao*[1]

[1]*School of Computer Science and Technology, Beijing Institute of Technology, Beijing 100081, People's Republic of China*
[2]*Key Laboratory of High Confidence Software Technologies, (Peking University) Ministry of Education, Beijing 100871, People's Republic of China*
[3]*Beijing Laboratory of Intelligent Information Technology, School of Computer Science and Technology Beijing Institute of Technology, Beijing 100081, People's Republic of China*
E-mail: liuhui2005@gmail.com

Abstract: Refactorings might be done using two different tactics: root canal refactoring and floss refactoring. Root canal refactoring is to set aside an extended period specially for refactoring. Floss refactoring is to interleave refactorings with other programming tasks. However, no large-scale case study on refactoring tactics is available. To this end, the authors carry out a case study to investigate the following research questions. (i) How often are root canal refactoring and floss refactoring employed, respectively? (ii) Are some kinds of refactorings more likely than others to be applied as floss refactorings or root canal refactorings? (iii) Do engineers employing both tactics have obvious bias to or against either of the tactics? They analyse the usage data information collected by Eclipse usage data collector. Results suggest that about 14% of refactorings are root canal refactorings. These findings reconfirm the hypothesis that, in general, floss refactoring is more common than root canal refactoring. The relative popularity of root canal refactoring, however, is much higher than expected. They also find that some kinds of refactorings are more likely than others to be performed as root canal refactorings. Results also suggest that engineers who have explored both tactics obviously tended towards root canal refactoring.

1 Introduction

1.1 Software refactoring and its tactics

Software refactoring is to restructure software internal structures to improve software quality, especially its maintainability, extensibility and reusability, whereas external behaviours of software applications are not changed [1, 2]. External behaviours of software applications include, but not limited to, software functionality. In some cases, other constrains, for example, no decrease in software performance, should be satisfied, as well. For example, refactorings applied to real-time applications should not decrease application performance. Otherwise, these applications might not respond as quickly as expected, which might result in system failure.

Software refactoring is widely used to facilitate software maintenance, and to delay the degradation effect of software aging. It is common that software applications are repeatedly modified because of evolving requirements, bug fixing etc. As a result, the source code shifts away from its original design structure, and it becomes difficult to read and extend the source code. To this end, software refactoring is employed to improve software applications' readability and extensibility.

Refactorings could be applied in two different tactics [3]. The first one is to set aside an extended period specially for refactoring. Murphy-Hill and Black [4] call these refactorings 'root canal refactorings'. Since the time slots are specially allocated for refactoring, these refactorings are interspersed with few non-refactoring activities. The other tactic is to apply refactorings in small bursts, and interleave refactorings heavily with other programming tasks, for example, functional development. Murphy-Hill and Black [4] call these refactorings 'floss refactoring'.

Root canal refactorings are different from floss refactorings, and the difference is important for researchers and tool vendors in this field. An overview of the difference is presented in Table 1. The most important aspects of the difference are discussed as follows.

First, smell detection tools are widely used to facilitate root canal refactorings, but they are rarely used to facilitate floss refactorings. When developers are carrying out root canal refactoring, they should identify refactoring opportunities throughout the whole application. However, manual identification of refactoring opportunities in large applications is difficult. To this end, experts have summarised a number of typical situations that may require software refactoring [5, 6]. Fowler calls these typical situations 'bad smells' [5, Chapter 3]. Researchers have proposed a number of detection algorithms to identify different kinds of code smells automatically or semi-automatically. For example, Kamiya *et al.* [7] propose a tool 'CCFinder' to detect clones in source code. Tsantalis and Chatzigeorgious [8] propose an approach to identifying feature envy. Bavota *et al.* [9, 10] propose an approach to identify extract class refactoring opportunities. Tsantalis and Chatzigeorgious [11] propose an approach to identifying

Table 1 Difference between refactoring tactics and its consequence

	Root canal refactoring	Floss refactoring
extended period for pure refactorings	yes	no
refactoring scope	large (e.g. the whole application)	small (e.g. a newly modified class)
smell detection tools	needed	not needed
number of refactoring opportunities found at a time	large	small
schedule algorithm	needed	not needed
interweaved with non-refactoring activities	no or slightly	heavily
seamless switch between refactoring and routine development	not needed	needed

refactoring opportunities to introduce polymorphism. Tourwe and Mens [12] propose a logic meta-programming approach to identify refactoring opportunities. Rompaey *et al.* [13] propose a metric-based approach to identify test smells and Munro [14] also proposes a metric-based approach to identify bad smells. By contrast, when developers are carrying out floss refactorings, they are usually confined to the newly added or modified methods or classes. Developers know these methods well, and they can find out refactoring opportunities manually and accurately.

Second, schedule of software refactorings could facilitate root canal refactorings, but floss refactoring might not need it. A large number of root canal refactoring opportunities might be found all at once by smell detection tools. According to our investigation [15, 16], carrying out these refactorings in different orders might result in different effect and requires different refactoring effort. Consequently, to maximise quality improvement and to minimise refactoring effort, developers should arrange application orders for all potential refactoring opportunities, which we call schedule of software refactorings [17, 18]. For a large number of root canal refactorings, developers might need the help of scheduling approaches [19]. By contrast, floss refactorings are carried out in small bursts, and only a small number of refactoring opportunities could be found at a time. As a result, no schedule is needed, or a simple schedule that could be managed manually would do.

Finally, refactoring tools for floss refactoring should support seamless switch between refactoring and routine development [4], whereas those for root canal refactoring do not. The reason is that floss refactorings are heavily interspersed with routine development activities. By contrast, root canal refactorings are carried out in time slots specially set aside for pure refactoring.

1.2 Code refactoring tactics

Floss refactoring is well recognised, and it has become one of the corner stones of software development. However, root canal refactoring is still in doubt. Fowler declares [5] that he is opposed to setting aside specific time for refactoring in almost all cases. Shore [20] also throw doubts on root canal refactoring. Murphy-Hill and Black [4] hold similar opinion, too.

On the other side, cases of root canal refactoring from the industry are frequently reported. For example, Microsoft usually reserves 20% of development efforts on thorough refactoring right after a software system is released and before the development of the next version of the system is initiated [21]. Consequently, we expect a large number of refactorings would be carried out as root canal refactorings by Microsoft. Our experiences in Newegg [http://www.newegg.com], a well-known international e-business company, where the first author once worked, also suggest that some might practice root canal refactoring frequently. Software development teams of the company develop and maintain an e-business system. After a new version of the system is released, development of the next version would not start immediately. Instead, an extended period would be allocated to evaluate the new release and to analyse new requirements. During this period, some software engineers are assigned to fixing bugs, but more are assigned to thoroughly restructuring (refactoring) the system. Software refactoring would improve 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 because of the improved extensibility because of root canal refactorings. 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 [22, 23].

The prosperity of the research in root canal refactoring might also suggest that root canal refactoring might be popular. As discussed in Section 1.1, bad smell detection and refactoring schedule are mainly used with root canal refactoring. Researchers have done lot of work in bad smell detection [6, 15] and refactoring schedule [16, 18], suggesting they might think highly of root canal refactoring. As declared in their papers, these tools that have been designed for root canal refactoring have been used by a large number of engineers.

Despite the debate, however, to the best of our knowledge there is no large-scale study on refactoring tactics. Murphy-Hill and Black [4] declared that floss refactoring is more popular than root canal refactoring. They analysed 40 Eclipse concurrent version system (CVS) commits, and found out that only 2% of refactorings were applied in root canal fashion [3]. To the best of our knowledge, these papers [3, 4] are the only two papers that investigate refactoring tactics. These papers are significant and interesting, but a larger scale case study is indispensable to make conclusions more convincing.

To this end, we conduct a case study to investigate the popularity of these tactics. In the case study, we analyse refactoring histories captured in real industry. Results suggest that about 14% of software refactorings are root canal refactoring, whereas others (86%) are floss refactoring. We observe that some kinds of refactorings are more likely than others to be done in root canal tactic. Results also suggest that engineers who have explored both tactics usually do much more root canal refactorings than floss refactorings.

Initial results of the investigation have been reported briefly in a short paper [24]. For space limitation, however, details of the investigation were not presented. In this paper, the following extensions have been made:

1. The investigation in this paper involves more data. The short paper [24] analysed data collected by Eclipse usage data collector (UDC) from January, 2010 to May, 2010. In this paper, the time window is expanded to August, 2010.

doi: 10.1049/iet-sen.2012.0121

As a result, the number of activities involved has increased from 615 788 499 to 985 261 598. In other words, the number of involved activities has increased by 60%.
2. An additional research question is investigated in this paper: do engineers who have explored both tactics have obvious bias to or against either of the tactics?
3. The accuracy of refactoring tactic identification is evaluated, and the results are presented in this paper (part D of Section 2.3.2).

The rest of the paper is structured as follows. Section 2 presents the analysis on refactoring histories captured by Eclipse UDC. Section 3 presents the results of the case study. Section 4 explains how threats to validity are coped, and Section 5 discusses implication of evaluation results. Section 6 presents a short overview of related work and Section 7 makes a conclusion.

2 Case study design

To investigate software refactoring tactics, we should collect refactoring history from a large number of software engineers. Otherwise, the investigation results could not be generalised. On one hand, it is challenging for us to collect such a large number of engineers for a controlled experiment. On the other hand, refactoring history captured by third parties from millions of engineers is publicly available, which making it is possible to conduct a case study on refactoring tactics. Consequently, we conduct a case study instead of an experiment.

In this section, we would introduce the design of the case study following the guidelines for reporting case studies in software engineering [25].

2.1 Research questions

The case study would investigate the following research questions:

'RQ1': How often are root canal refactoring and floss refactoring employed, respectively? As discussed in Section 1.1, some technologies (tools) are used only for a specific refactoring tactic. For example, code smell detection tools are mainly designed for root canal refactoring, and they are rarely used for floss refactoring. Consequently, the popularity of each tactic is important for tool vendors and researchers in this field to decide which refactoring tactic to support. Although Fowler [5] and Shore [20] throw doubts on root canal refactoring, researchers have proposed a lot of approaches and tools for bad smell detection [6, 15] and refactoring schedule [16, 18] that are mainly used with root canal refactoring. Consequently, investing the popularity of these two refactoring tactics with large refactoring history might help researchers to align their research with what actually happens in practice.

'RQ2': are some kinds of refactorings, for example, 'extract method' and 'move method', more likely than others to be done in root canal fashion or floss refactoring fashion? For those who are developing algorithms and tools used for root canal refactoring only, they would like to pick up those kinds of refactorings that are more likely than others to be done as root canal refactoring. The same is true for those developing tools for floss refactoring only.

'RQ3': do engineers employing root canal refactoring and floss refactoring have obvious bias to or against either of the tactics? Only if you have tried both of them, you know which one is more suitable to you. Consequently, tool vendors are also interested in the relative popularity of the tactics among those who know and have tried both tactics.

Besides the motivations listed above, answering these research questions might help to give researchers and engineers an overview of the state-of-practice.

2.2 Data collection

2.2.1 Challenges in discovering refactorings and distinguishing refactoring tactics: There are four potential approaches to capturing or discovering refactoring histories that serve as a basic for the evaluation of the three research questions in Section 2.1, but each of them has its weakness and strength [26]. We would give them a brief introduction focusing on the challenges in distinguishing root canal refactorings from floss refactorings.

Observing programmers manually: The first approach to capturing refactorings is to observe programmers manually [26]. It is the most direct way to capture software refactorings. Moreover, with this approach it is easy to tell refactoring tactics that are indispensable to the investigation of all of the research questions listed in Section 2.1. However, a large-scale manual observation is too expensive for academic researchers. Moreover, observers might misunderstand observed programmers.

Analysing code histories: The second approach is to analyse code histories, that is, compare different versions of the same software system [27, 28]. The comparison could be done manually or automatically with tool support. If characteristics of each refactorings are explicitly specified, it is possible to automatically recover changes between different versions that can be expressed as refactorings. In this way, numerous refactorings can be recovered because most open source applications have their code histories publicly available online. One weakness of this approach is that the identification is inaccuracy, especially in identifying complex refactorings without distinguished characteristics.

Another weakness of the approach is that the accurate (or relative) happening time of refactorings could not be recovered. This happening time is critical for the analysis of refactoring tactics. Consequently, it is challenging to distinguish tactics for refactorings found by analysing code histories.

Mining commit logs: The third approach is to mine commit logs. This approach assumes that when programmers commit changes they would explicitly specify in commit logs whether any refactorings have been performed, and if yes which refactorings. However, it has been reported [26, 29] that in most cases the assumption does not hold.

One of the advantages of this approach is that we can infer how refactoring activities are interspersed with other activities by analysing commit logs. It is would make the identification of refactoring tactics possible. Identification of refactoring tactics is the basic for the investigation of research questions RQ1–RQ3.

Mining logs of refactoring tools: The fourth approach to identifying refactorings is to mine logs of refactoring tools. Once programmers carry out refactorings with tool support, these tools (or monitors installed as plug-ins) will write logs indicating when, where and which refactorings are carried out. These logs are periodically and automatically sent to a central server. This approach is less expensive because no human intervention is required. Moreover, it enables us to capture large number of refactoring activities from numerous programmers all over the world, which is not

easy to accomplish by observing programmers manually. One weakness of this approach is that refactorings carried out manually cannot be identified correctly. Manual refactorings might not activate refactoring tools, and thus they would not be logged by refactoring tools.

It is challenging to distinguish refactoring tactics from logs of refactoring tools, whereas identification of refactoring tactics is critical for all of the research questions under investigation. The reason is that we do not know what happens during the interval of successive refactoring activities. During this period, the engineering might carry out routine development, for example, adding new function or just leave the workbench (for rest or other tasks). This makes it challenging to infer how refactoring activities are interspersed with other activities. It in turn hinders the identification of refactoring tactics.

Eclipse UDC, employed in this paper, is an extension of this approach. UDC captures all activities on Eclipse including execution of refactoring commands. Consequently, with data collected by Eclipse UDC, we can discover not only which refactorings have been performed, but also how these refactoring activities are interspersed with other activities. It is critical for refactoring tactic analysis that is one of the main targets of this paper.

Eclipse itself, besides UDC plug-in, also captures refactoring histories. The histories, as shown in a screenshot from Eclipse in Fig. 1, contain even more details than UDC. As a result, these histories are potential data sources for research on how refactorings are performed. However, they are not adopted in our research, because these histories are stored in local disks owned by programmers instead of a central server. As a result, to obtain enough data for analysis, we have to request these data from numerous programmers one by one.

According to our analysis that is presented in preceding paragraphs, a brief summary of the strengths and weaknesses of the four approaches is presented in Table 2.

The first approach (observing programmers manually) is not chosen because it is too expensive. The second one (analysing code histories) is not chosen because the accurate happening time of refactorings could not be recovered, and thus refactoring tactics could not be discovered. The third one (mining commit logs) is not chosen because of poor performance [26, 29].

2.2.2 UDC and data collection: Eclipse UDC [http://www.eclipse.org/org/usagedata] is an Eclipse plug-in developed and maintained by the Eclipse foundation. The UDC collects information about how individuals use Eclipse. This information is automatically sent to servers hosted by the Eclipse foundation. The intent of the UDC is to use this data to better understand how developers are using Eclipse.

The usage data records what is being used and when (timestamp). Recorded events include: (i) load of bundles (e.g. activation of an Eclipse plug-in); (ii) invocation of commands (e.g. invocation of command 'copy' by pressing keys 'Ctrl' and 'C'); (iii) activation of actions embedded in Eclipse menus or toolbars (e.g. click on the 'Run' button); (iv) state change of perspectives (e.g. activation or close of the Eclipse resource perspective); (v) state change of views (e.g. activation of the Eclipse outline view); and (viii) state change of editors (e.g. activation of the Eclipse Java editor).

All refactoring commands integrated in 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: '1000008, 'started, bundle', org.eclipse.equinox.simpleconfigurator, 1.0.200.v20100503, org.eclipse.jdt.ui.JavaPerspective, 1333072648742l' The seven fields of an UDC record are presented and explained in Table 3.

Records captured between January, 2010 and August 2010 are publicly available online [http://www.archive.eclipse.org/technology/phoenix/usagedata/] for academic research, and our investigation is done on these records. At the period, Eclipse UDC captured more than 985 261 598 activities performed by more than 1 054 713 users worldwide (the actual number of users might be a bit smaller than that of UserIDs because a UserID captured by UDC represents a

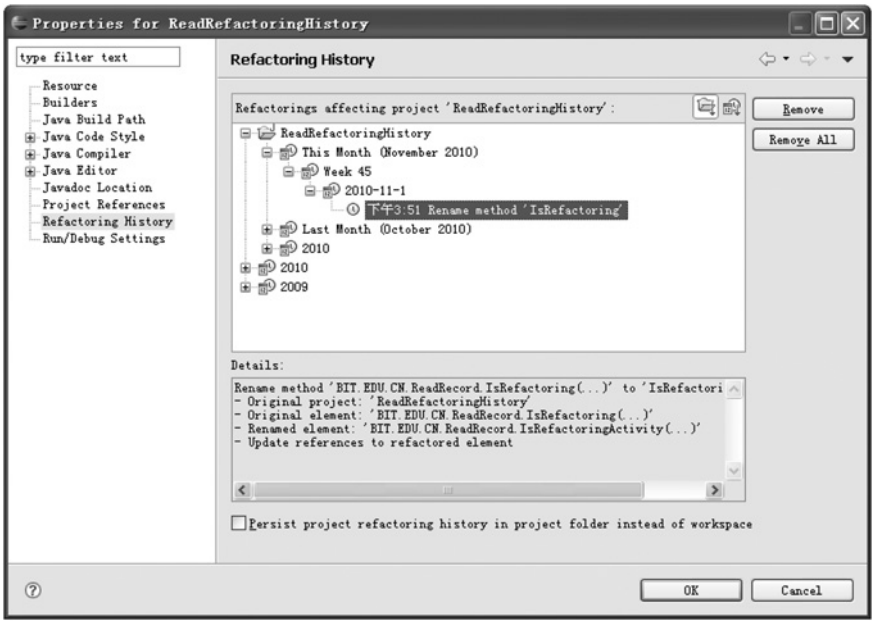

Fig. 1 *Screenshot from eclipse refactoring history*

doi: 10.1049/iet-sen.2012.0121

Table 2 Comparison among different approaches to discovering refactorings

	Observing programmers manually	Analysing code histories	Mining commit logs	Mining logs of refactoring tools
scalability	poor	good	good	good
automated refactorings only	no	no	no	yes
accuracy	good	fair	poor	fair
distinguishing refactoring tactics	easy	hard	fair	fair

unique workstation and workspace, and some users might work on multiple machines and workspace).

2.3 Analysis procedure

2.3.1 Identification of refactorings: If a refactoring command embedded in Eclipse is activated, that is, a refactoring is carried out by the user, a record would be sent to UDC. The record should fill its fields 'description, what' and 'kind' with the refactoring ID, constant string 'executed' and string 'command', respectively. The key to the identification of refactorings is the field 'description' which is filled with the ID of the executed refactoring command.

For each refactoring on the refactoring menu of Eclipse, we activated it and checked the record captured by UDC. In this way, the IDs for these refactorings were collected. Eclipse also provides some refactoring commands (application programming interface (APIs)) that might be used by third-party plug-ins. Once these commands are activated, the UDC would generate a record as well. The manager of UDC, manually picked up IDs corresponding to these refactoring commands. The full list of IDs is presented in Appendix, and all records that contain any of these IDs would be classified as refactorings.

It should be noted that if refactorings are not carried out by refactoring commands embedded in Eclipse, these refactoring activities could not be identified. It might introduce some threats to validity (discussed in Section 4).

To date, more than 90 code refactorings have been proposed [http://www.refactoring.com/catalog/index.html]. However, not all of them are supported by Eclipse. Eclipse selectively implemented the most popular ones. Similar strategies of selective implementation are adopted by other refactoring tools, for example, ReSharper [http://www.jetbrains.com/resharper/].

Table 3 Fields of an UDC record

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 1000008
what	indicates what happens. For the sample record, a bundle was 'started'.
kind	indicates what kind of bundle is involved, for example, this bundle is a 'command', a 'view', or an 'editor'.
bundleId	the ID of the involved bundle (plug-in). For the sample record, the bundle ID is 'org. eclipse. equinox. Simple configurator'
bundle version	version of the bundle. For the sample record, the version is 1.0.200.v20100503
description	provides some information, for example, ID of the source bundle, about the event. 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 workstation. For the sample case, the time stamp '1333072648742l' suggests that this even happened at 2012-03-30 09:57:28.742

2.3.2 Identification of root canal refactoring: *Overview:* 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.
(2) Second, if a number of successive subsequences of activities performed by the same user form an extended working period (longer than a half working day), 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, that is, the user is carrying out root canal refactoring at the period.

Non-refactoring activities within pure refactoring days: 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. The reasons are explained as follows:

- First, while carrying out refactorings, software engineers might explore the views of Eclipse, for example, '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, for example, '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.

These activities are not always done as part of software refactoring, and thus they cannot be simply classified as refactoring activities. In most cases, these activities, for example, regression testing and bug fixing, happen in routine development where no refactorings are carried out.

Since of these activities, we cannot expect the UDC to capture pure refactoring activities even though developers

are carrying out root canal refactoring. However, we might expect a great ratio of refactoring activities (to all activities captured) compared with that of floss refactoring. This is the basis for the identification of root canal refactorings.

Threshold estimation for the minimal refactoring density: To distinguish root canal refactorings from floss refactorings with refactoring density (the ratio of refactoring activities to all activities), we should estimate the minimal refactoring density when an extended period is specifically set aside for root canal refactoring, and validate that it does work. To this end, we conduct the following pilot.

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

All refactorings were performed on Eclipse, and UDC was 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.

Once refactorings were completed, we manually checked the revision made to the projects to make sure that no functional change had been done, that is, the whole period had been completely set aside for refactoring. After that, we calculated the ratio of refactoring activities to all activities. The ratios for the three engineers 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 were captured by UDC as non-refactoring activities. The reasons have been discussed in Section 2.3.2. We adopt the mean value 1% as the minimal refactoring density.

Accuracy of refactoring tactic identification: To evaluate the accuracy of the identification, we conducted another pilot. Two graduate students (with more than 2 years of professional working experience with software development before they came back to school for master degrees) were selected for the evaluation. They were developing a unified modeling language (UML) modelling tool. We asked one of them (PA) to carry out root canal refactorings and the other (PB) to develop in XP style for two days. Their activities were monitored by UDC. After that, data captured by UDC were analysed. At ideal conditions, all refactorings carried out by PA should be identified as root canal refactoring activities, whereas all refactoring activities carried out by PB should be identified as floss refactorings.

With the threshold of 1%, 86% of refactoring activities carried out by PA are identified as root canal refactoring activities, whereas all refactoring activities carried out by PB are identified as floss refactorings. In other words, the approach, together with the threshold, leads to a recall of 86% and a precision of 100%. The high precision is not surprising because it is observed that the ratio of refactoring activities to all activities (0.22%) of PB is much smaller than the threshold (1%).

However, we also observe that about 14% of root canal refactoring activities are not identified correctly. The reason is that the density of refactoring activities varies from time to time (non-uniform distribution). Some refactoring activities are more intensively interspersed with non-refactoring activities, for example, exploring projects, switching among different views and text edits. As a result, the density at the period is lower than the threshold, which leads to false negatives in the identification.

3 Results and findings

Statistics of the analysed data, as well as analysis results, are presented in Table 4. The data analysed includes 985 261 598 activities from 1 054 713 users. Among them, 1 019 268 activities are refactorings and 138 995 are root canal refactorings. Detailed analysis of the statistics is presented and discussed as follows.

3.1 Relative popularity of the tactics (answering research question RQ1)

Of about 138 995 out of 1 019 268 refactoring activities are root canal refactorings. The ratio of root canal refactorings to all refactorings ('CRatio' for short in the rest of this paper) is 138 995/1 019 268 = 13.64%. In other words, about 86% of refactorings are floss refactorings. The result is consistent with the declaration that floss refactoring is more common than root canal refactoring [3].

However, the 'CRatio' (13.64%) is much higher than that (2%) reported in [3]. 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 the investigation reported in [3], only 40 commits of Eclipse are analysed to discover refactoring tactics. In contrast, our investigation involve 985 261 598 activities performed by millions of engineers worldwide working on different projects. Another possible reason is that Murphy-Hill *et al.* [3] distinguished refactoring tactics manually, whereas we did it automatically. Manual identification might be more accurate in small data although it is hard to apply manual identification to large data.

3.2 Variation over different kinds of refactorings (answering research qsuestion RQ2)

For different kinds of refactorings, engineers might take different tactics. Consequently, 'CRatio' might vary among different kinds of refactorings. We calculate 'CRatio' for each kind of software refactoring. Results are presented on

Table 4 Statistics of the analysed data and analysis results

Items	Value
number of involved users	1 054 713
number of activities	985 261 598
number of refactorings	1 019 268
ratio of refactorings to all activities	0.1%
number of floss re factorings	880 273
ratio of floss re factorings to all re factorings	86.36%
number of root Canal re factorings	138 995
ratio of root canal re factorings to All re factorings ('CRatio')	13.64%
ratio of floss refactorings to all re factorings ('FRatio')	86.36%
number of users applying refactorings	244 186
number of users applying floss refactoring only	237 828
number of users taking both tactics	4812
number of users applying root canal refactoring only	1546

doi: 10.1049/iet-sen.2012.0121

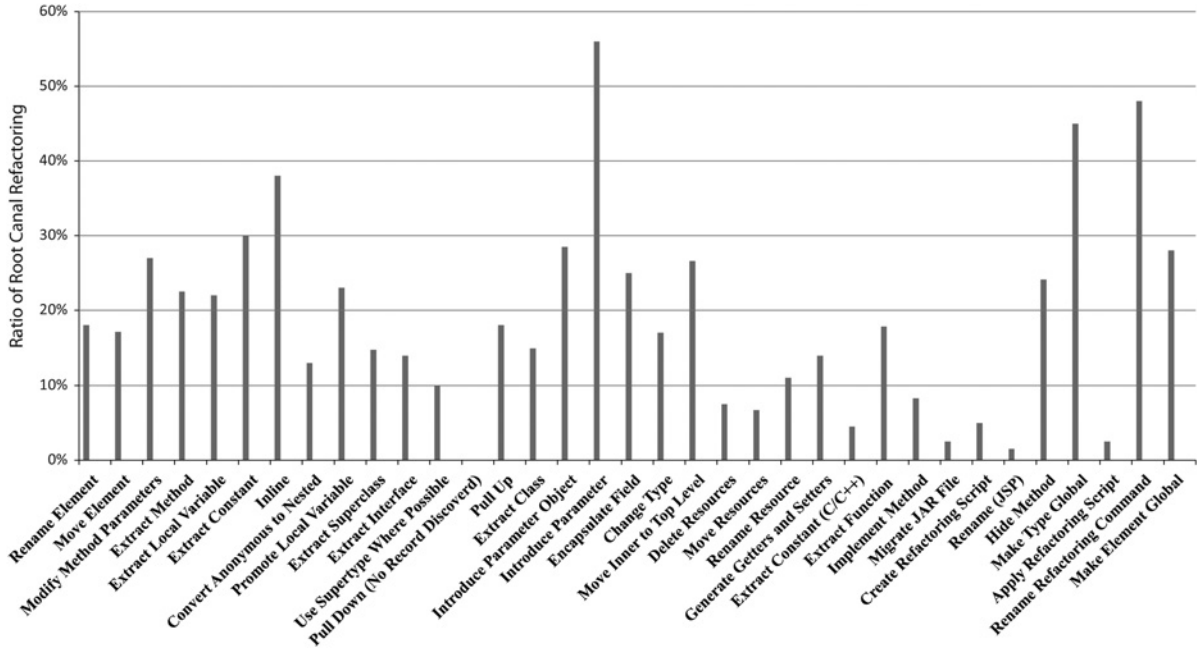

Fig. 2 *Variation of CRatio over different kinds of refactorings*

Fig. 2. We also calculate the ratio of floss refactorings to all activities ('FRatio') for each kind of software refactoring, and results are presented in Fig. 3.

From this figure, we observe that 'CRatio' varies dramatically from 1.59 to 50% (excluding those not captured by UDC whose ratios of refactorings are unavailable). The results suggest that some kinds of refactorings (those with grater 'CRatio'), for example, 'introduce parameter' and 'inline', are more likely than others to be carried out as root canal refactoring. In contrast, some kinds of refactorings (those with smaller 'CRatio'), for example, 'extract class' and 'use super type', are more likely than others to be done as 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.

3.3 Variation over different engineers (answering research question RQ3)

To answer research question 'RQ3', we calculate 'CRatio' for each of the involved software engineers. If 'CRatio' is zero, the corresponding engineer applies floss refactoring only. If 'CRatio' is 100%, the corresponding engineer applies root canal refactoring only. Otherwise ('CRatio' ranges between zero to 100%), the corresponding engineer takes both of the tactics, and the proportion between the tactics could be deduced from 'CRatio'.

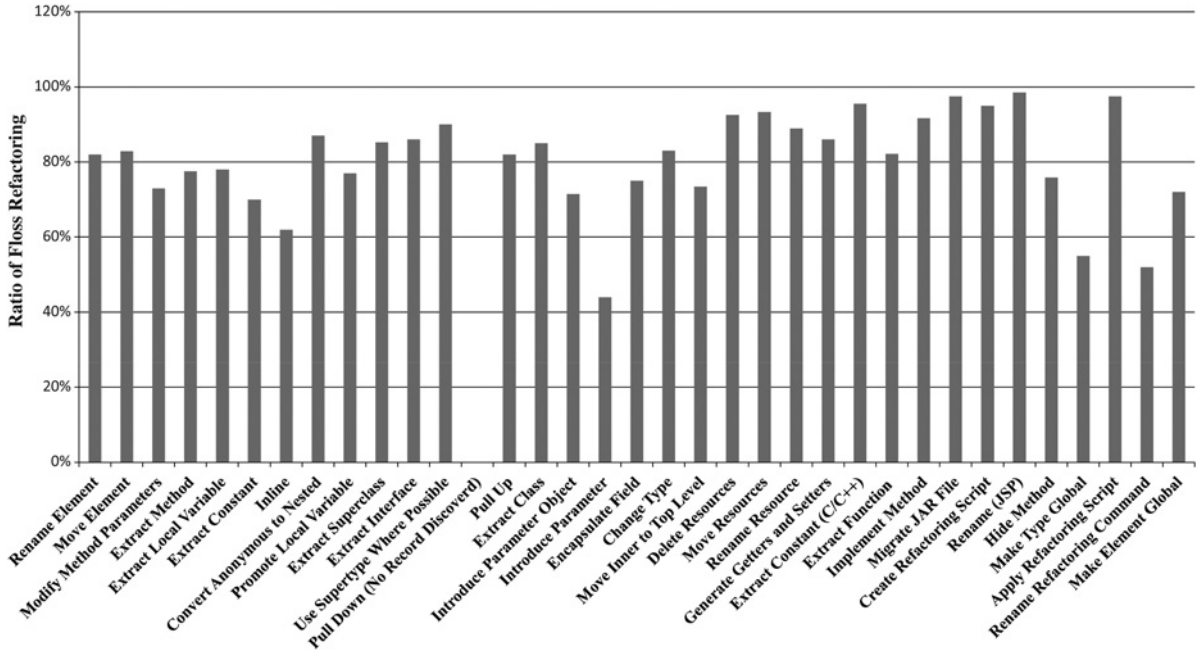

Fig. 3 *Variation of FRatio over different kinds of refactorings*

Table 5 Distribution of 'CRatio' among engineers employing both tactics

Range of CRatio, %	Number of users
0–10	5
10–20	50
20–30	180
30–40	310
40–50	500
50–60	520
60–70	663
70–80	800
80–90	1004
90–100	780

Some of the results are presented in Tables 4 and 5. From these tables, we observe that most engineers (237 828/244 186 = 97%) applied floss refactoring only. We also observe that more than six thousand engineers applied root canal refactorings (sum of the last two rows in Table 4): 4812 engineers employed both tactics (the last but one row), and 1546 engineers applied root canal refactorings only (the last row).

For the 4812 engineers employing both refactoring tactics, the distribution of 'CRatio' is presented in Table 5. From this table, we observe that there is an obvious bias to root canal refactoring: most of them (approximately 80%) have done more root canal refactorings than floss refactorings ('CRatio' > 50%).

We have discovered 129 382 refactorings applied by these engineers who employed both of the refactoring tactics. Among these refactoring, 91 088 are root canal refactorings (91 088/129 382 = 70.4%). Consequently, it is not surprising that most of them achieved a high 'CRatio'.

4 Threats to validity

4.1 Limited to eclipse and UDC

The first threat to external validity is that all software engineers involved in UDC data analysis (Section 2) work on Eclipse. Consequently, conclusions drawn on these engineers might not be true for others working on Visual Studio or other Integrated Development Environments (IDEs). However, these development environments, for example, Visual Studio, are similar to Eclipse. Visual Studio provides a list of refactoring menus similar to those provided by Eclipse.

All the involved users have installed Eclipse UDC and agreed to upload their activity records. UDC is not distributed together with Eclipse Classic [There are many versions of Eclipse, and some of them might contain UDC.]. Consequently, it should be installed manually before any activity can be captured by UDC server. In our experiences, however, most software engineers do not know Eclipse UDC, let alone installation. Consequently, it is possible that the involved users have some specific characters that other engineers do not have, and these specific characters might lead to the findings.

4.2 Small-scale threshold estimation

Another threat to external validity exists in threshold estimation in Section 2.3.2. The estimation is done by measuring activities of three engineers. The number of involved engineers is relatively small, which might throw doubts on the generalisability of the threshold. To reduce the threat, we conducted a case study in Section 2.3.2 to validate whether the threshold could be used for root canal refactoring identification. Results suggest that the recall and precision are 86 and 100%, respectively.

4.3 Reliability of refactoring identification

The identification of refactoring activities from other activities captured by UDC (Section 2) might be inaccurate. The reasons are explained as follows.

First, the proposed approach might miss some refactorings. Since the approach identifies refactorings by checking invocation history of refactoring commands, refactorings that have been carried out manually or with the help of third-party refactoring tools, would be missed because they do not rely on the invocation of the Eclipse refactoring commands.

Second, some activities performed as part of refactorings might be captured as non-refactoring activities by the proposed approach. Details have been presented in Section 2.

4.4 Reliability of refactoring classification

In Section 2.3.2, root canal refactorings are identified by ratio of refactoring activities to all activities. In other words, dense refactoring activities in an extended period (longer than 4 h) are identified as root canal refactoring activities. However, because different development activities (and refactorings) might take different effort, the density alone might be inaccurate in distinguishing root canal refactorings from floss refactorings. To validate the accuracy of the approach, we have conducted a small case study in Section 2.3.2, and the results suggest that its recall and precision are 86 and 100%, respectively.

5 Implications of the results

First of all, the results might encourage researchers to reevaluate the usefulness of their research on root canal refactoring technologies, for example, code smell detection algorithms. As discussed in Section 1.2, root canal refactoring has been doubted by experts, and an initial investigation reported that it is rarely applied, that is, less than 2% of refactorings are applied in this fashion. As a result, related research on code smell detection and refactoring scheduling that is associated with root canal refactoring is also put into doubt. However, the results of our case study suggest that about 14% of refactorings are carried out in root canal fashion. Although these results also suggest that root canal refactoring is not as common as floss refactoring, the ratio of root canal refactorings (to all refactorings) found in this paper (14%) is much greater than that (2%) found previously [3]. Our results also suggest that for some kinds of refactorings, for example, 'introduce parameter', root canal refactoring is even more popular than floss refactoring. Consequently, the results reported in this paper might give researchers a chance to reconsider their work on root canal refactoring.

Second, researchers who are designing code smell detection algorithms might focus on those smells that are more likely to be resolved in root canal fashion. As discussed in Section 1, code smell detection tools are usually associated with root canal refactoring instead of floss refactoring. Considering that some kinds of

doi: 10.1049/iet-sen.2012.0121

refactorings (e.g. 'inline, introduce parameter, extract constant' and 'make type global') have much higher likelihood of being applied in root canal fashion, we might suggest researchers who are designing code smell detection algorithms to focus on smells (e.g. 'short methods, unnecessary dependency, static constants' and 'private types') that would lead to these refactorings.

Third, researchers who are designing refactoring tools for floss refactoring [4] could focus on those refactorings that are more likely than others to be applied in floss fashion. According to Murphy-Hill and Black's [4] research, different refactoring tactics may have quite different requirements on refactoring tool support, and existing refactorings tools are rarely designed for floss refactoring. The results reported in this paper might suggest which kinds of refactoring tools should be restructured first for floss refactoring.

Fourth, complex refactoring scheduling approaches [17] specially designed for root canal refactoring might be simplified by removing those refactorings (or the corresponding code smells) that are rarely applied in root canal fashion. Some researchers are also designing scheduling approaches for a single kind of code smells, for example, the scheduling approach for clone resolution [19]. These researchers might also benefit from the results reported in this paper by focusing on those code smells holding the highest likelihood of being resolved by root canal refactorings.

6 Related work

A series of significant papers in this field have been published by Murphy-Hill and his colleagues. Murphy-Hill and Black [4] 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 case study.

Murphy-Hill *et al.* [3] 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. Although multiple data sources (refactoring activities captured by Maylar [30] from 41 volunteer programmers, activities captured by Eclipse UDC, refactoring histories from four developers and Eclipse CVS) are analysed in their paper, however, this conclusion is validated on only 40 Eclipse CVS commits and other data sources are analysed for other purposes. To make the validation more convincing, we validate it on 985 261 598 activities captured by UDC from 753 367 users worldwide. Moreover, we also make a further investigation on root canal refactoring, and some interesting findings are reported. Details of these findings are presented in Section 3.

Murphy-Hill and Black [26] list and discuss four methods for refactoring identification: mining the commit log, analysing 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.* [30] 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 1 054 713 software engineers.

Xing and Stroulia [31] report an Eclipse case study investigating refactoring practice and its tool support. They identify refactorings via UMLDiff [32] 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 [32] 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 [31].

7 Conclusions and future work

Although people have argued for quite a long time on refactoring tactics, to the best of our knowledge, no large-scale investigation of refactoring tactics has been publicly reported. To this end, we analyse activities captured by Eclipse UDC to discover the popularity of different refactoring tactics. By analysing 985 261 598 activities from 1 054 713 engineers, we obtain some interesting findings. First, root canal refactoring is not as common as floss refactoring, but its popularity is much higher than ever reported. Second, some kinds of refactorings have higher likelihood than others to be applied in root canal fashion. Finally, engineers employing both tactics tend obviously towards root canal refactoring.

The investigation is valuable because different refactoring tactics might have different requirements on refactoring technologies and supporting tools. The findings reported in this paper might give some help to researcher in this field in selecting their research subjects and setting research goals.

Further investigation should involve more data from the industry that are captured or discovered by more than one approach. It should be interesting, as well, to investigate why some kinds of refactorings are more likely to be carried out as root canal refactorings. A future investigation might discover the commonness among those refactorings that are the most likely to be applied in root canal refactoring. In this paper, we have found that different kinds of refactorings have quite different likelihood of being applied in root canal/floss fashion. However, we have not yet investigated the reasons. The reason is that we cannot contact anonymous users applying these refactorings, and we cannot access the involved source code, either.

It is also interesting to investigate why some engineers apply more floss refactorings than root canal refactorings, whereas others carry out more root canal refactoring than floss refactoring. Possible factor might include, but not limited to, engineer' personality, project character and the strategy of the company. Future investigation should discover which factors might have non-trivial impact on the selection of refactoring tactics.

Future work is also needed to investigate why engineers employing both tactics tend obviously towards root canal refactoring. It would be interesting to investigate how many of the engineers who apply floss refactoring only do not know root canal refactoring at all. It is also interesting to know how many of them would change their refactoring tactics once they know both tactics.

8 Acknowledgments

The authors would like to say thanks to the anonymous reviewers of COMPSAC'12 for their constructive comments and suggestions on how to revise and extend the original version. We also thank the editors and reviewers of this paper for their valuable comments and suggestions. The work is funded by the National Natural Science Foundation of China (nos. 61003065, 61272169, 61272361) and Specialised Research Fund for the Doctoral Program of Higher Education (no. 20101101120027).

9 References

1 Mens, T., Touwe, T.: 'A survey of software refactoring', *IEEE Trans. Softw. Eng.*, 2004, **30**, (2), pp. 126–139
2 Opdyke, W.F.: 'Refactoring object-oriented frameworks'. PhD thesis, University of Illinois at Urbana-Champaign, 1992
3 Murphy-Hill, E., Parnin, C., Black, A.P.: 'How we refactor, and how we know it'. Proc. 31st Int. Conf. Software Engineering, 2009, pp. 287–297
4 Murphy-Hill, E., Black, A.P.: 'Refactoring tools: fitness for purpose', *IEEE Softw.*, 2008, **25**, (5), pp. 38–44
5 Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D.: 'Refactoring: improving the design of existing code' (Addison Wesley Professional, 1999)
6 Wake, W.C.: 'Refactoring workbook' (Addison Wesley, 2003)
7 Kamiya, T., Kusumoto, S., Inoue, K.: 'CC Finder: a multi-linguistic token based code clone detection system for large scale source code', *IEEE Trans. Softw. Eng.*, 2002, **28**, (6), pp. 654–670
8 Tsantalis, N., Chatzigeorgiou, A.: 'Identification of move method refactoring opportunities', *IEEE Trans. Softw. Eng.*, 2009, **99**, (RapidPosts), pp. 347–367
9 Bavota, G., De Lucia, A., Oliveto, R.: 'Identifying extract class refactoring opportunities using structural and semantic cohesion measures', *J. Syst. Softw.*, 2011, **84**, (3), pp. 397–414
10 Bavota, G., Oliveto, R., De Lucia, A., Antoniol, G., Gueheneuc, Y.: 'Playing with refactoring: Identifying extract class opportunities through game theory'. 2010 IEEE Int. Conf. Software Maintenance (ICSM)., September 2010, pp. 1–5
11 Tsantalis, N., Chatzigeorgiou, A.: 'Identification of refactoring opportunities introducing polymorphism', *J. Syst. Softw.*, 2010, **83**, (3), pp. 391–404
12 Tourwe, T., Mens, T.: 'Identifying refactoring opportunities using logic meta programming'. Proc. Seventh European Conf. Software Maintenance And Reengineering (CSMR'03), 2003, pp. 91–100
13 Van Rompaey, B., Du Bois, B., Demeyer, S., Rieger, M.: 'On the detection of test smells: A metrics-based approach for general fixture and eager test', *IEEE Trans. Softw. Eng.*, 2007, **33**, (12), pp. 800–817
14 Munro, M.J.: 'Product metrics for automatic identification of 'bad smell' design problems in java source-code'. Proc. 11th IEEE Int. Symp. Software Metrics, 2005, September 2005, pp. 15–15
15 Liu, H., Ma, Z., Shao, W., Niu, Z.: 'Schedule of bad smell detection and resolution: a new way to save effort', *IEEE Trans. Softw. Eng.*, 2012, **38**, (1), pp. 220–235
16 Liu, H., Li, G., Ma, Z., Shao, W.: 'Conflict aware scheduling of software refactorings', *IET Softw.*, 2008, **2**, (5), pp. 446–460
17 Liu, H., Li, G., Ma, Z., Shao, W.: 'Scheduling of conflicting refactorings to promote quality improvement'. Proc. 22nd IEEE/ACM Int. Conf. Automated Software Engineering (ASE'07), 2007, pp. 489–492
18 Liu, H., Yang, L., Niu, Z., Ma, Z., Shao, W.: 'Facilitating software refactoring with appropriate resolution order of bad smells'. Proc. Seventh Joint Meeting of the European Software Engineering Conf. (ESEC) and the ACM SIGSOFT Symp. Foundations of Software Engineering (ESEC/FSE'09), 2009, pp. 265–268
19 Bouktif, S., Antoniol, G., Merlo, E., Neteler, M.: 'A novel approach to optimize clone refactoring activity'. in Mike, C., (ed.), Proc. Eighth Annual Conf. Genetic and Evolutionary Computation (GECCO'06), July 2006, pp. 1885–1892. ACM SIGEVO (formerly ISGEC), ACM Press
20 Shore, J.: 'Design debt'. Software Profitability Newsletter, 01 February 2004. Available at http://www.jamesshore.com/Articles
21 Cusumano, M.A., Selby, R.W.: 'Microsoft secrets' (Free Press, 1995)
22 Stroulia, E., Kapoor, R.: 'Metrics of refactoring-based development: An experience report'. Proc. Seventh Int. Conf. Object-Oriented Information System, 2001, pp. 113–C122
23 Weber, R., Helfenberger, T., Keller, R.K.: 'Fit for change: Steps towards effective software maintenance'. Proc. 21st IEEE Int. Conf. Software Maintenance – Industrial and Tool volume, 2005, pp. 26–33. Recipient of Best Industrial Paper Award
24 Liu, H., Gao, Y., Niu, Z.: 'An initial study on refactoring tactics'. IEEE 36th Int. Conf. Computer Software and Applications, 2012, pp. 213–218
25 Runeson, P., Höst, M.: 'Guidelines for conducting and reporting case study research in software engineering', *Empir. Softw. Eng.*, 2009, **14**, (2), pp. 131–164
26 Murphy-Hill, E., Black, A.P., Danny, D., Parnin, C.: 'Gathering refactoring data: a comparison of four methods'. Proc. Second Workshop on Refactoring Tools (WRT'08), 2008, pp. 7:1–7:5
27 Dig, D., Comertoglu, C., Marinov, D., Johnson, R.: 'Automated detection of refactorings in evolving components'. Proc. 20th European Conf. Object-Oriented Programming (ECOOP'06), Nantes, France, 3–7 July 2006, (LNCS vol. 4067), pp. 404–42. Springer Berlin/Heidelberg
28 Peter, W., Diehl, S.: 'Are refactorings less error-prone than other changes?'. Proc. 2006 Int. Workshop on Mining software repositories (MSR'06), 2006, pp. 112–118
29 Parnin, C., Görg, C.: 'Improving change descriptions with change contexts'. Proc. 2008 Int. Working Conf. Mining Software Repositories (MSR'08), 2008, pp. 51–60
30 Murphy, G.C., Kersten, M., Findlater, L.: 'How are java software developers using the eclipse ide?', *IEEE Softw.*, 2006, **23**, pp. 76–83
31 Xing, Z., Stroulia, E.: 'Refactoring practice: How it is and how it should be supported – an eclipse case study'. IEEE Int. Conf. Software Maintenance, 2006, pp. 458–468
32 Xing, Z., Stroulia, E.: 'Umldiff: an algorithm for object-oriented design differencing'. Proc. 20th IEEE/ACM Int. Conf. Automated Software Engineering, 2005, pp. 54–65. Xing, Zhenchang and Stroulia, Eleni

10 Appendix

The UDC IDs corresponding to refactoring activities in Eclipse.

See Table 6.

doi: 10.1049/iet-sen.2012.0121

Table 6 Refactorings and their corresponding IDs captured by UDC

ID	Refactoring
java.rename.element	rename
java.move.element	move
java.modify.method.parameters	change parameter
java.extract.method	extract method
java.extract.local.variable	extract local variable
java.extract.constant	extract constant
java.Inline	inline
java.convert.anonymous.to.nested	convert anonymous to nested
java.promote.local.variable	promote local variable
java.Extract.superclass	extract superclass
java.extract.interface	extract interface
java.use.supertype	use supertype
java.pull.down	pull down
java.pull.up	pull up
java.extract.class	extract class
java.introduce.parameter.object	introduce parameter object
java.introduce.parameter	introduce parameter
java.self.encapsulate.field	encapsulate field
java.change.type	change type
java.move.inner.to.top.level	move inter to top level
org.eclipse.ltk.ui.refactoring.commands.deleteResources	delete resources
org.eclipse.ltk.ui.refactoring.commands.moveResources	move resources
org.eclipse.ltk.ui.refactoring.commands.renameResource	rename resources
org.eclipse.cdt.ui.refactor.getters.and.setters	generate getters and setters
org.eclipse.cdt.ui.refactor.extract.constant	extract constant (C/C++)
org.eclipse.cdt.ui.refactor.extract.function	extract function
org.eclipse.cdt.ui.refactor.implement.method	implement method
org.eclipse.jdt.ui.refactor.migrate.jar	migrate a JAR file
org.eclipse.ltk.ui.refactor.create.refactoring.script	create refactoring script
org.eclipse.jst.jsp.ui.refactor.rename	rename JSP elements
org.eclipse.cdt.ui.refactor.hide.method	hide method
org.eclipse.wst.xsd.ui.refactor.makeTypeGlobal	make type global
org.eclipse.ltk.ui.refactor.apply.refactoring.script	apply refactoring script
org.eclipse.photran.ui.renamerefactoringcommand	rename refactoring command
org. eclipse. wst. xsd. ui. refactor. make Element Global	make element global