The Need for Richer Refactoring Usage Data

Mohsen Vakilian Nicholas Chen Stas Negara Balaji Ambresh Rajkumar
Roshanak Zilouchian Moghaddam Ralph E. Johnson

University of Illinois at Urbana-Champaign

{mvakili2, nchen, snegara2, rajkuma1, rzilouc2, rjohnson}@illinois.edu

Abstract

Even though modern Integrated Development Environments (IDEs) support many refactorings, studies suggest that automated refactorings are used infrequently, and few developers use anything beyond Rename and Extract refactorings. Little is known about *why* automated refactorings are seldom used. We present a list of challenging questions whose answers are crucial for understanding the usability issues of refactoring tools. This paper argues that the existing data sources—Eclipse UDC, Eclipse refactoring histories, version control histories, etc.—are inadequate for answering these questions. Finally, we introduce our tools to collect richer usage data that will enable us to answer some of the open research questions about the usability of refactoring tools. Findings from our data will foster the design of the next generation of refactoring tools.

Categories and Subject Descriptors D.2.6 [*Software Engineering*]: Programming Environments; D.2.0 [*Software Engineering*]: General

General Terms Experimentation, Human Factors, Measurement

Keywords Usability, Refactoring, Automated Refactoring Tools, User Study

1. Introduction

Refactoring is a kind of program transformation that improves the internal design of a program without changing its external behavior. Opdyke identified several recurring refactorings two decades ago [13]. Today, software development processes such as Extreme Programming (XP) prescribe refactoring and modern Integrated Development Environments (IDEs) support many refactorings. Automated tools have been designed to simplify the task of applying common refactorings [15]. However, recent studies suggest that developers are not taking full advantage of these tools and Rename and Extract refactorings constitute most uses of automated refactorings [8, 12].

Why do not developers use the automated refactorings more? Perhaps programmers just need more training in the tools, the tools implement the wrong refactorings, or most of the refactorings are not done frequently enough to justify their automation. Is the problem that the tools have poor user interfaces? Perhaps they are based on a flawed understanding of how developers use these tools in their workflows.

The answers to these questions have deep implications for designers as they develop new refactoring tools. Answering questions about the usability of refactoring tools is difficult because they span many different aspects of the design of the tools. In this paper, we argue that more detailed data than what are currently available are required to answer these questions. We anticipate that some of the questions cannot be fully answered by solely relying on the usage data. Therefore, we need to interview programmers to better understand their perception of the refactoring tools and the problems that they find with these tools. Richer refactoring usage data will also enable us to ask more specific questions during the interviews and get more accurate answers.

Almost all we know about the usability of refactoring tools comes from the studies based on a few *coarse-grained* and *inconsistent* sets of data (See Section 2). There is a large amount of coarse-grained data, but, only a small amount of fine-grained data from a few developers on the usage of refactoring tools. These data sets are inconsistent and difficult to correlate because they capture different data over different time intervals from different developers. Others have reported some interesting statistics on the usage of refactoring tools based on these data [12]. However, we need to study more aspects of automated refactorings such as their configurations, previews, and failures to discover the major usability problems of automated refactorings.

The limitations of the existing data have motivated us to develop our own usage data collectors, *CodingSpectator* and *CodingTracker* [1], for capturing *richer* data about high-level refactorings and low-level code edits. We will combine

This paper was published in the Proceedings of the Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU) at the ACM Onward! and SPLASH Conferences. October, 2011. Portland, Oregon, USA.

the data of these two data collectors and our interviews with developers to answer many open research questions about the usability of refactoring tools (See Section 3).

In this paper, we first give an overview of the existing data on the usage of refactoring tools (See Section 2). Then, we discuss some of the current findings about the usability of refactoring tools and present several new open research questions (See Section 3). Answers to these questions will give us a better understanding of the major usability problems of mainstream refactoring tools. However, we show that the existing data sets are not sufficient to answer these research questsions. So, we will present our technique for collecting richer and more consistent data about automated and manual refactorings (See Section 4). Our richer usage data and interviews will make it possible to answer some of the open questions that are vital to design the next generation of refactoring tools that better align with how developers work (See Section 6).

2. Limitations of Existing Data

Other researchers have analyzed multiple sets of data to study the usability of refactoring tools. The generalizability and number of conclusions that can be drawn by these studies depend on the quality and quantity of these data sets. This section discusses the strengths and weaknesses of each set of refactoring usage data that others have studied to understand the usability properties of refactoring tools.

2.1 Mylyn Monitor

Mylyn Monitor was the first usage data collector tool for Eclipse [8]. It captured data about the use of various features and plug-ins of Eclipse. Mylyn monitor captured the interaction history of each user. The interaction history included the time of invoking commands and changes to views, perspectives, and editor selections. A subset of the interaction history captured information about the refactoring commands. Mylyn Monitor collected data from 41 developers. Since its focus was not refactorings, it did not collect detailed information about how programmers interacted with automated refactorings, e.g. the configurations of automated refactorings.

2.2 Aggregated Eclipse Usage Data

Most releases of Eclipse come with a plug-in called the Eclipse *Usage Data Collector (UDC)* [2]. The intent of UDC is to help the developers of Eclipse better understand how the users utilize different features of it. UDC records various events including the usage of all perspectives, views, and commands in Eclipse. It also records invocations of refactorings as commands, and captures the ID and the time of invocation of every command locally. Then, it regularly uploads the collected data to the Eclipse foundation servers, if the user agrees to share his or her data.

The Eclipse foundation aggregates and publishes the data every year. The aggregated data reports the total number of invocations of each command by all users during every month. UDC has been in operation since April 2008. So far, the Eclipse foundation has published the aggregated data until January 2010. Figure 1 shows a sample of the aggregated Eclipse usage data publicly available on the UDC website.

Since UDC is pre-installed in most releases of Eclipse and captures coarse-grained data that do not contain sensitive information, many users agree to submit their data. On average, UDC has received data from 168,100 users per month. Even though not all users of UDC use Eclipse for Java programming, a significant fraction of them invoke Java specific commands.

The UDC data record what automated refactorings get invoked. However, they do not contain more detailed information about how an automated refactoring is performed, e.g. how the user has configured the tool. In addition, there is no one-to-one mapping between the IDs of the commands of the UDC data and the refactoring IDs. For instance, the UDC data do not distinguish the Rename Local Variable refactoring from the Rename Method refactoring. The UDC data distinguish the six variants of the Extract refactoring but not the ten, four and three variants of the Rename, Move and Inline refactorings, respectively. The impacts of these refactorings depend on the program elements on which they are invoked. For example, the impact of renaming a local variable is limited to a method while renaming a method could affect multiple files. Since the impacts of these refactorings are different, the users might use them differently. Therefore, it is useful to differentiate the refactorings performed on different kinds of program elements to study how programmers use such refactorings.

2.3 Time-stamped Eclipse Usage Data

The Eclipse UDC plug-in captures the time of invocation of every command locally [2]. So far, the Eclipse foundation has published the time-stamped usage data from January 2009 until August 2010. The UDC set of data with timestamps reports the exact time rather than the year and month of invocation of every command. The time-stamped usage data make it possible to study the refactoring activities in a small window of time. Figure 2 shows a sample of the time-stamped Eclipse usage data from our own local system.

2.4 Eclipse Refactoring Histories

Eclipse logs completed refactorings locally. The intent of this kind of log is to replay the refactorings. Eclipse represents every refactoring operation as a *refactoring descriptor*. Refactoring descriptors make it possible to decouple the back-end refactoring engine from the front-end user interface of refactoring tools. A refactoring descriptor contains enough information to replay the refactoring on the same source code. In addition to the ID and the time of invocation, a refactoring descriptor stores all the parameters and configurations provided on the user interface of the refactor-

```
yearmonth,command,bundleId,bundleVersion,executeCount,userCount
200901,org.eclipse.jdt.ui.edit.text.java.inline,org.eclipse.jdt.ui,3.4.0.v20080603-2000,68,38
200901,org.eclipse.jdt.ui.edit.text.java.extract.local.variable,org.eclipse.jdt.ui,3.4.0.v20080603-2000,1252,436
...
```

Figure 1. Sample of the data collected in aggregated Eclipse usage data

```
what,kind,bundleId,bundleVersion,description,time
executed,command,org.eclipse.jdt.ui,3.7.0.201107172337,"org.eclipse.jdt.ui.edit.text.java.inline",1312611465324
opened,editor,org.eclipse.jdt.ui,3.7.0.201107172337,"org.eclipse.jdt.ui.CompilationUnitEditor",1312611648385
...
```

Figure 2. Sample of the data collected in the time-stamped Eclipse usage data

```
<?xml version="1.0" encoding="UTF-8"?>
<session version="1.0">
 <refactoring comment="Extract method 'private static
  void print()' from 'MyClass.main()' to 'MyClass'
  - Original project: 'MyClass'
  - Method name: 'print'
  - Destination type: 'MyClass'
  - Declared visibility: 'private'"
  comments="false" destination="0" exceptions="false"
  description="Extract method 'print'" flags="786434"
  id="org.eclipse.jdt.ui.extract.method"
  input="/src<{MyClass.java" name="print"
  replace="false" selection="68 37" visibility="2"
  stamp="1313173552362" version="1.0"/>
  ...
</session>
```

Figure 3. Sample of the data collected in Eclipse refactoring histories

ing tool. Figure 3 shows a sample of the data collected in the Eclipse refactoring histories.

When the user performs an automated refactoring, Eclipse creates a refactoring descriptor and logs it in its local refactoring history. If the user undoes the automated refactoring later, Eclipse will remove the corresponding refactoring descriptor from the refactoring history.

The Eclipse refactoring histories are more detailed than the UDC data, but these two data sets are not consistent (See Sections 2.2, 2.3). The UDC data capture every refactoring command that is initiated, while refactoring histories only capture the refactorings that have been completed and have not been later undone.

Since the Eclipse refactoring histories have been designed to replay refactorings, they do not capture information about the failures of automated refactorings and the error messages that they report. However, a better understanding of the failures of refactoring tools might lead to improvements in the usability of such tools [10].

Although the Eclipse refactoring histories are richer than UDC data, there has been no systematic mechanism for collecting the Eclipse refactoring histories to a central repository, and few refactoring histories are available. Robbes has reported on the usage of refactoring tools by himself and another developer [14]. Four developers who primarily maintain the automated refactorings of Eclipse have shared their refactoring histories with several researchers. Additionally, eight developers of the Eclipse Mylyn project used to check their refactoring histories in their CVS repository. Nonetheless, Eclipse started to capture some of the refactorings only in the middle of the data collection from these developers [12]. As a result, the refactoring histories of these 12 developers are incomplete because the data about some of the early automated refactorings are missing.

2.5 Version Control Histories

Version control systems contain the source code of many open source projects. Several researchers have studied the histories of version control systems for refactoring activities [6, 7, 12, 17]. Finding the refactorings between two versions of the source code requires sampling, metrics, and heuristics. The following are some of the disadvantages of relying on the version control histories for studying refactorings:

1. Developers may not commit all their refactorings to version control systems.
2. Sometimes, there are different ways to refactor one version of the code to another. In such cases, it is often challenging to determine which sequence of refactorings have led to a particular version of the source code.
3. Most of the time, it is almost impossible to tell whether a refactoring has been performed manually or through a tool by just comparing two snapshots of the source code. Therefore, some have tried to match the version control commits with Eclipse refactoring histories to distinguish the automated and manual refactorings [12]. However, few developers have made their refactoring histories available (See Section 2.4).

3. Research Questions

Others have raised interesting research questions about the usability of refactoring tools and discussed differenet methods for studying the use of such tools [8, 11]. This section categorizes and discusses these research questions and introduces a few new ones. We briefly discuss the existing work and possible implications of answers to each set of ques-

tions. These open questions require richer data about the usage of refactoring tools beyond what are currently available. However, even our richer usage data will not be sufficient to answer all of our questions. For example, we may not capture enough data about automated refactorings that programmers rarely use. Therefore, we will conduct interviews and combine quantitative and qualitative approaches to gain deeper insight [5]. Table 1 lists the different categories from this section and summarizes whether existing data sources can be used to answer the questions.

3.1 Manual and Automated Refactorings

Manual refactorings are believed to be error-prone and automated refactorings are supposed to help developers perform refactorings quickly and correctly [16]. Nonetheless, a study of the Eclipse refactoring histories and version control histories of two Eclipse development teams suggested that automated refactorings are underused, that is, developers opt to perform most refactorings manually despite the availability of automated support. Prior research has also suggested that programmers intersperse refactorings with other edits [12].

Answers to the following questions will provide a better understanding of how programmers combine automated refactorings with manual changes and the characteristics of automated refactorings that are underused. This understanding may have implications for designing tools that are used more frequently and better match the workflow of programmers.

1. How do programmers intersperse refactorings with other kinds of program changes?
2. How common are simple and complex automated refactorings?
3. How frequent are refactorings?
4. Are automated refactorings underused?
5. How often are different refactorings performed with and without tools?

Eclipse refactoring and version control histories cannot be used to answer the above questions reliably (See Sections 2.4 and 2.5). We need to collect data from more developers about the fine-grained edits, and not just coarse-grained commits of version control systems. Capturing fine-grained edits also allows us to find out how the use of automated refactorings differ for intermediate refactorings and the final refactorings found in consecutive snapshots of a version control system.

3.2 Awareness

A survey conducted on a few developers suggested that developers do not use some automated refactorings because they are unaware of the tools [12]. Our interviews with four students of the software engineering course at the University of Illinois corroborate this observation. Even though the students had received training on several refactorings they were unaware of the automated support for many others. More quantitative and qualitative data are required to answer the following questions:

1. What refactorings do developers frequently underuse because of their unawareness?
2. How do developers become aware of certain automated refactorings?

Not all questions about the awareness of automated refactorings can be answered based on the usage data. We can capture data only about automated refactorings that programmers are aware of. More interviews with developers are required to better understand the effect of awareness on the use of automated refactorings.

3.3 Methods of Invocation

There are different ways to invoke a refactoring operation. A programmer may invoke the refactoring by selecting a program element in the editor or a structured view. Also, the programmer could use the *Quick Assist* feature of Eclipse to get a list of applicable refactorings on the current selection. Quick Assist narrows down the decision space of programmers by proposing only a subset of refactorings applicable to the current context. Other methods of invoking refactorings have also been proposed [9]. However, little is known about the impact of each invocation method on the use of automated refactorings [8]. It is not clear if programmers prefer to invoke refactorings from within editors or graphical representations of code. Also, we do not know if light-weight methods of invocations such as Quick Assist influence programmers to use automated refactorings more often.

Some automated refactorings operate on specific program elements and are unavailable on others. Others have observed that it is sometimes tricky to select a valid piece of code for invoking the Extract Method refactoring [10]. However, it is still unclear how prevalent the selection problems are for all automated refactorings. Based on these observations, we find it essential to seek answers to the following questions in order to invent better ways of invoking the automated refactorings:

1. What refactorings are more difficult to invoke? What refactorings programmers fail to invoke because of wrong input selections?
2. How do developers prefer to select the input program elements of automated refactorings? Do they prefer textual or structural selections?
3. How often do developers use the Quick Assist feature of Eclipse to invoke automated refactorings? Do programmers use the autoamted refactorings supported by Quick Assist more often?

Research Aspects	Mylyn Monitor	UDC Data		Refactoring Histories	Refactoring and Version Control Histories
		Aggregated	Time-stamped		
Manual and automated refactorings (Section 3.1)	⬙	⬙	⬙	⬙	⬙
Awareness (Section 3.2)	⬙	⬙	⬙	⬙	⬙
Methods of invocation (Section 3.3)	⬙	◇	⬙	◇	◇
Configuration of automated refactorings (Section 3.4)	◇	◇	◇	⬙	◇
Previewing the outcomes of automated refactorings (Section 3.5)	◇	◇	◇	◇	◇
Exceptional cases (Section 3.6)	◇	◇	◇	◇	◇
Cancellations and undos (Section 3.7)	⬙	◇	⬙	◇	◇

Table 1. Can we use the existing data sources to help answer our research questions? ◆(Yes); ⬙(Partially); ◇(No)

3.4 Configuration of Automated Refactorings

Most automated refactorings are configurable. For instance, the wizard of the Extract Method refactoring in Eclipse provides options to configure the access modifier, declared exceptions, and comment of the new method. Knowing what options developers frequently set and ignore helps tool designers streamline the user interface.

Others have analyzed the Eclipse refactoring histories (See Section 2.4) of two development teams to understand how often these developers have changed the default configurations of their tools [12]. These results have raised the following new set of questions that cannot be answered using any of the existing sets of refactoring usage data (See Section 2).

1. Do programmers manually change the outcomes of automated refactorings? Do these changes indicate any options that are missing from the user interfaces of automated refactorings?
2. Under what circumstances do developers change the default options of their tools? Can the automated refactorings suggest better default options based on the recent developers' activities in the IDE?
3. How long do developers spend on configuring various automated refactorings? Are long configuration times indicators of usability problems?

3.5 Previewing the Outcomes of Automated Refactorings

Most automated refactorings offer the option of previewing their proposed changes. Typically, for each file that will be affected by the refactoring, the preview dialog shows two versions: the one before applying the refactoring and the one after.

One developer raised concerns about the usability of the preview dialog in a survey [12]. None of the existing data sets contain data about previews of refactorings and we do not know of any empirical research on this aspect of refactoring tools. We collect data about the preview actions of developers to answer the following questions (See Section 4):

1. How often do programmers preview automated refactorings?
2. What refactorings are programmers more likely to preview?

3.6 Exceptional Cases

Automated refactorings try to guarantee that their changes will be behavior-preserving. If the tool detects that a particular use of it may change the observed behavior of the program, it reports some error messages to the developer. The automated refactorings of Eclipse rate the severity levels of their messages as *information*, *warning*, *error*, and *fatal error*. Eclipse prevents the user from continuing an automated refactoring only if it has found a fatal error.

Others have observed that the Extract Method refactoring of Eclipse communicates error messages to its user poorly [10]. By collecting more data about the error messages reported by automated refactorings (See Section 4), we will be able to answer interesting questions about the usability of many refactorings including the following:

1. What automated refactorings present error messages more frequently?
2. What are the most frequent error messages about?

3. How do error messages affect the developer's future uses of the tool?

3.7 Cancellations and Undos

Others have found undo and erase events to be indicators of usability problems in creation oriented applications like Google SketchUp [3, 4]. Similarly, we hypothesize that cancellations and undos of automated refactorings may be indicators of usability problem. However, none of the existing data sets capture such events and we are unaware of any research study on such refactoring actions. Collecting data about cancellations and undos of automated refactoring refactorings could shed light on the following questions:

1. What are the most highly canceled and undone automated refactorings?
2. Does the number of cancellations and undos correlate with other indicators of complexity such as complicated error messages, change impact of the refactoring, and the amount of time spent on configuring the automated refactoring?
3. Do the error messages of automated refactorings influence developers to cancel or undo their refactorings?

4. CodingSpectator and CodingTracker

To address the limitations of existing data sources (Section 2) and to answer the open research questions about the usability of refactoring tools (Section 3), we have developed our own minimally intrusive usage data collection tools: CodingSpectator and CodingTracker. Both tools modify Eclipse to capture more data. So far, 26 developers have been using our data collectors in real-world settings. Based on the feedback from our participants, we have iteratively improved the user experience and data collection capabilities of CodingSpectator and CodingTracker so that they do not slow down or crash Eclipse or interfere with developers' work.

The following is a sample of the data that CodingSpectator collects about refactorings:

1. Whether the refactoring was performed, canceled or unavailable
2. The kind of refactoring (Rename Local Variable, Move Static Method, Extract Method, etc.)
3. When the refactoring was invoked
4. The kind of selection that was made to invoke the refactoring (textual or structural)
5. The selected source code element
6. Whether the refactoring was invoked using Quick Assist
7. How long the user spent on each step of the refactoring (configuration, preview, error message comprehension)
8. How the user has configured the refactoring
9. Any problems reported by the refactoring tool

CodingTracker captures fine-grained edit operations down to the level of an insertion or deletion of a character in the Java editors of Eclipse. It captures the edits so precisely that it can later replay them to show the code changes in action. In addition, CodingTracker collects other data including the undone refactorings and commits to version control systems.

CodingSpectator and CodingTracker gather consistent data that we will combine for our future analyses. Our rich sets of usage data and interviews allow us to answer all the open research questions we presented in Section 3. However, capturing rich usage data from real programming environments brings challenges because it raises privacy issues and makes it difficult to recruit participants.

5. Related Work

Murphy et al.'s study of 74 developers using the Java tools in Eclipse [8] stimulated research in this area. Their study was the first to report the usage frequencies of automated refactorings. Our study will extend their work by collecting more data to explain the reported usage patterns. For example, we collect the selections used to invoke the automated refactorings in order to find the potential problems in invoking automated refactorings.

Murphy-Hill et al. then analyzed the existing data from Murphy et al., the Eclipse foundation [2] and two other data sources to study how developers refactor [12]. Their work was the first to suggest that programmers underuse automated refactorings. They also surveyed five developers to get some insight about why developers may not use automated refactorings. Our work builds on theirs and collects richer quantitative and qualitative data to better understand *why* programmers underuse automated refactorings. For example, we collect data about failures of automated refactorings to sutdy how they affect the user experience.

In another study, Murphy-Hill et al. instructed several participants to apply the Extract Method refactoring on several open source projects. They observed that their participants had difficulty in selecting the right pieces of code to extract. Based on this observation, they proposed improvements to the user interface of automated refactorings [10]. Our study extends theirs by detecting problems in invoking refactorings and comprehending error messages in real-world environments for many more refactorings.

Mealy et al. developed a collection of usability requirements for refactoring tools based on existing general usability standards. Our work complements theirs by collecting usage data that may reveal some of the usability problems of refactoring tools.

Akers et al. instrumented Google SketchUp [3], a 3-D modeling application, and recorded invocations of undo and erase events [4]. They were able to identify several usability problems by analyzing the undo and erase events. Their work exemplifies the potential of monitoring for identifying

usability problems. Inspired by their work, we are collecting detailed information about the usage of refactorings such as undoing, cancellation, termination, and unavailability to identify potential usability problems.

6. Future Work

We have been actively recruiting developers to use our data collectors and send us data. We plan to analyze the collected data to identify interesting patterns in the usage of refactoring tools and answer some of the open research questions on the usability of these tools. Also, we plan to interview some of our participants to get a better insight about the specific usage patterns. Finally, we will use our usage data and interviews to derive design implications for building the next generation of developer-oriented refactoring tools.

References

[1] CodingSpectator and CodingTracker. http://codingspectator.cs.illinois.edu.

[2] Eclipse Usage Data. http://www.eclipse.org/epp/usagedata.

[3] Google SketchUp. http://sketchup.google.com.

[4] D. Akers, M. Simpson, R. Jeffries, and T. Winograd. Undo and Erase Events as Indicators of Usability Problems. In *Proceedings of the 27th International Conference on Human Factors in Computing Systems*, CHI '09, pages 659–668, New York, NY, USA, 2009. ACM.

[5] J. Creswell. *Research Design: Qualitative, Quantitative, and Mixed Methods Approaches*. Sage Publications, 2009.

[6] S. Demeyer, S. Ducasse, and O. Nierstrasz. Finding Refactorings via Change Metrics. In *Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '00, pages 166–177, New York, NY, USA, 2000. ACM.

[7] M. Kim, D. Cai, and S. Kim. An Empirical Investigation into the Role of API-level Refactorings during Software Evolution. In *Proceeding of the 33rd International Conference on Software Engineering*, ICSE '11, pages 151–160, New York, NY, USA, 2011. ACM.

[8] G. C. Murphy, M. Kersten, and L. Findlater. How Are Java Software Developers Using the Eclipse IDE? *IEEE Software*, 23:76–83, July 2006.

[9] E. Murphy-Hill, M. Ayazifar, N. Carolina, and A. P. Black. Restructuring Software with Gestures. In *Visual Langauges and Human-Centric Computing*, 2011.

[10] 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, pages 421–430, New York, NY, USA, 2008. ACM.

[11] 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, pages 287–297, Washington, DC, USA, 2009. IEEE Computer Society.

[12] E. Murphy-Hill, C. Parnin, and A. P. Black. How We Refactor, and How We Know It. *IEEE Transactions on Software Engineering*, 99(PrePrints), 2011.

[13] W. F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, Champaign, IL, USA, 1992. UMI Order No. GAX93-05645.

[14] R. Robbes. Mining a Change-Based Software Repository. In *Proceedings of the Fourth International Workshop on Mining Software Repositories*, MSR '07, page 15, Washington, DC, USA, 2007. IEEE Computer Society.

[15] D. Roberts, J. Brant, and R. Johnson. A Refactoring Tool for Smalltalk. *Theory and Practice of Object Systems*, 3:253–263, October 1997.

[16] P. Weißgerber, S. Diehl, and C. Görg. Mining Refactorings in ARGOUML. In *Proceedings of the 2006 International Workshop on Mining Software Repositories*, MSR '06, pages 175–176, New York, NY, USA, 2006. ACM.

[17] Z. Xing and E. Stroulia. Refactoring Practice: How it is and How it Should be Supported - An Eclipse Case Study. In *Proceedings of the 22nd IEEE International Conference on Software Maintenance*, pages 458–468, Washington, DC, USA, 2006. IEEE Computer Society.