How We Refactor, and How We Know It

Emerson Murphy-Hill
Portland State University
emerson@cs.pdx.edu

Chris Parnin
Georgia Institute of Technology
chris.parnin@gatech.edu

Andrew P. Black
Portland State University
black@cs.pdx.edu

Abstract

Much of what we know about how programmers refactor in the wild is based on studies that examine just a few software projects. Researchers have rarely taken the time to replicate these studies in other contexts or to examine the assumptions on which they are based. To help put refactoring research on a sound scientific basis, we draw conclusions using four data sets spanning more than 13 000 developers, 240 000 tool-assisted refactorings, 2500 developer hours, and 3400 version control commits. Using these data, we cast doubt on several previously stated assumptions about how programmers refactor, while validating others. For example, we find that programmers frequently do not *indicate refactoring activity in commit logs, which contradicts assumptions made by several previous researchers. In contrast, we were able to confirm the assumption that programmers do frequently intersperse refactoring with other program changes. By confirming assumptions and replicating studies made by other researchers, we can have greater confidence that those researchers' conclusions are generalizable.*

1. Introduction

Refactoring is the process of changing the structure of a program without changing the way that it behaves. In his book on the subject, Fowler catalogs 72 different refactorings, ranging from localized changes such as EXTRACT LOCAL VARIABLE, to more global changes such as EXTRACT CLASS [5]. Based on his experience, Fowler claims that refactoring produces significant benefits: it can help programmers add functionality, fix bugs, and understand software [5, pp. 55–57]. Indeed, case studies have demonstrated that refactoring is a common practice [19] and can improve code metrics [1].

However, conclusions drawn from a single case study may not hold in general. Studies that investigate a phenomenon using a single research method also may not hold. To see why, let's look at one particular example that uses a single research method: Weißgerber and Diehl's study of 3 open source projects [18]. Their research method was to apply a tool to the version history of each project to detect high-level refactorings such as RENAME METHOD and MOVE CLASS. Low- and medium-level refactorings, such as RENAME LOCAL VARIABLE and EXTRACT METHOD, were classified as *non*-refactoring code changes. One of their findings was that, on every day on which refactoring took place, non-refactoring code changes also took place. What we can learn from this depends on the relative frequency of high-level and mid-to-low-level refactorings. If the latter are scarce, we can infer that refactorings and changes to the projects' functionality are usually interleaved at a fine granularity. However, if mid-to-low-level refactorings are common, then we cannot draw this inference from Weißgerber and Diehl's data alone.

In general, validating conclusions drawn from an individual study involves both replicating the study in wider contexts and exploring factors that previous authors may not have explored. In this paper we use both of these methods to confirm — and cast doubt on — several conclusions that have been published in the refactoring literature.

Our experimental method takes data from four different sources (described in Section 2) and applies several different refactoring-detection strategies to them. We use this data to test nine hypotheses about refactoring. The contributions of our work lie in both the experimental method used when testing these hypotheses, and in the conclusions that we are able to draw:

- The RENAME refactoring tool is used much more frequently by ordinary programmers than by the developers of refactoring tools (Section 3.1);
- about 40% of refactorings performed using a tool occur in batches (Section 3.2);
- about 90% of configuration defaults of refactoring tools remain unchanged when programmers use the tools (Section 3.3);
- messages written by programmers in commit logs do not reliably indicate the presence of refactoring (Section 3.4);

- programmers frequently *floss refactor*, that is, they interleave refactoring with other types of programming activity (Section 3.5);
- about half of refactorings are not high-level, so refactoring detection tools that look exclusively for high-level refactorings will not detect them (Section 3.6);
- refactorings are performed frequently (Section 3.7);
- almost 90% of refactorings are performed manually, without the help of tools (Section 3.8); and
- the kind of refactoring performed with tools differs from the kind performed manually (Section 3.9).

In Section 4 we discuss the interaction between these conclusions and the assumptions and conclusions of other researchers.

2. The Data that We Analyzed

The work described in this paper is based on four sets of data. The first set we will call *Users*; it was originally collected in the latter half of 2005 by Murphy and colleagues [7] who used the Mylyn Monitor tool to capture and analyze fine-grained usage data from 41 volunteer programmers in the wild using the Eclipse development environment (http://eclipse.org). These data capture an average of 66 hours of development time per programmer; about 95 percent of the programmers wrote in Java. The data include information on which Eclipse commands were executed, and at what time. Murphy and colleagues originally used these data to characterize the way programmers used Eclipse, including a coarse-grained analysis of which refactoring tools were used most often. Murphy-Hill and Black have also used these data as a source of evidence for the claim that refactoring tools are underused [10].

The second set of data we will call *Everyone*; it is publicly available from the Eclipse Usage Collector [16], and includes data from every user of the Eclipse Ganymede release who consented to an automated request to send the data back to the Eclipse Foundation. These data aggregate activity from over 13 000 Java developers between April 2008 and January 2009, but also include non-Java developers. The data count how many programmers have used each Eclipse command, including refactoring commands, and how many times each command was executed. We know of no other research that has used these data for characterizing programmer behavior.

The third set of data we will call *Toolsmiths*; it includes refactoring histories from 4 developers who primarily maintain Eclipse's refactoring tools. These data include detailed histories of which refactorings were executed, when they were performed, and with what configuration parameters. These data include all the information necessary to recreate the usage of a refactoring tool, assuming that the original source code is also available. These data were collected between December 2005 and August 2007, although the date ranges are different for each developer. This data set is not publicly available and has not previously been described in the literature. The only study that we know of using similar data was published by Robbes [14]; it reports on refactoring tool usage by Robbes himself and one other developer.

The fourth set of data we will call *Eclipse CVS*, because it is the version history of the Eclipse and JUnit (http://junit.org) code bases as extracted from their Concurrent Versioning System (CVS) repositories. Commonly, CVS data must be preprocessed before analysis. Unfortunately, CVS does not maintain records showing which file revisions were committed as a single transaction. The standard approach for recovering transactions is to find revisions committed by the same developer with the same commit message within a small time window [20]; we use a 60 second time window. Henceforth, we use the word "revision" to refer to a particular version of a file, and the word "commit" to refer to one of these commit transactions. We excluded from our sample (a) commits to CVS branches, which would have complicated our analysis, and (b) commits that did not include a change to a Java file.

In our experiments, we focus on a subset of the commits in *Eclipse CVS*. Specifically, we randomly sampled from about 3400 source file commits (Section 3.4) that correspond to the same time period, the same projects, and the same developers represented in *Toolsmiths*. Using these data, two of the authors (Murphy-Hill and Parnin) inferred which refactorings were performed by comparing adjacent commits manually. While many authors have mined software repositories automatically for refactorings (for example, Weißgerber and Diehl [18]), we know of no other research that compares refactoring tool logs with code histories.

3. Findings on Refactoring Behavior

In this section we analyze these four sets of data and discuss our findings.

3.1. Toolsmiths and Users Differ

We hypothesize that the refactoring behavior of the programmers who develop the Eclipse refactoring tools differs from that of the programmers who use them. Toleman and Welsh assume a variant of this hypothesis — that the designers of software tools erroneously consider themselves typical tool users — and argue that the usability of software tools should be objectively evaluated [17]. However, as far as we know, no previous research has tested this hypothesis, at least not in the context of refactoring tools. To

Refactoring Tool	Toolsmiths				Users				Everyone	
	Uses	Use %	Batched	Batched %	Uses	Use %	Batched	Batched %	Uses	Use %
Rename	670	28.7%	283	42.2%	1862	61.5%	1009	54.2%	179871	74.8%
Extract Local Variable	568	24.4%	127	22.4%	322	10.6%	106	32.9%	13523	5.6%
Inline	349	15.0%	132	37.8%	137	4.5%	52	38.0%	4102	1.7%
Extract Method	280	12.0%	28	10.0%	259	8.6%	57	22.0%	10581	4.4%
Move	147	6.3%	50	34.0%	171	5.6%	98	57.3%	13208	5.5%
Change Method Signature	93	4.0%	26	28.0%	55	1.8%	20	36.4%	4764	2.0%
Convert Local To Field	92	3.9%	12	13.0%	27	0.9%	10	37.0%	1603	0.7%
Introduce Parameter	41	1.8%	20	48.8%	16	0.5%	11	68.8%	416	0.2%
Extract Constant	22	0.9%	6	27.3%	81	2.7%	48	59.3%	3363	1.4%
Convert Anonymous To Nested	18	0.8%	0	0.0%	19	0.6%	7	36.8%	269	0.1%
Move Member Type to New File	15	0.6%	0	0.0%	12	0.4%	5	41.7%	838	0.3%
Pull Up	12	0.5%	0	0.0%	36	1.2%	4	11.1%	1134	0.5%
Encapsulate Field	11	0.5%	8	72.7%	4	0.1%	2	50.0%	1739	0.7%
Extract Interface	2	0.1%	0	0.0%	15	0.5%	0	0.0%	1612	0.7%
Generalize Declared Type	2	0.1%	0	0.0%	4	0.1%	2	50.0%	173	0.1%
Push Down	1	0.0%	0	0.0%	1	0.0%	0	0.0%	279	0.1%
Infer Generic Type Arguments	0	0.0%	0	-	3	0.1%	0	0.0%	703	0.3%
Use Supertype Where Possible	0	0.0%	0	-	2	0.1%	0	0.0%	143	0.1%
Introduce Factory	0	0.0%	0	-	1	0.0%	0	0.0%	121	0.1%
Extract Superclass	7	0.3%	0	0.0%	*	-	*	*	558	0.2%
Extract Class	1	0.0%	0	0.0%	*	-	*	*	983	0.4%
Introduce Parameter Object	0	0.0%	0	-	*	-	*	*	208	0.1%
Introduce Indirection	0	0.0%	0	-	*	-	*	*	145	0.1%
Total	2331	100%	692	29.7%	3027	100%	1431	47.3%	240336	100%

Table 1. Refactoring tool usage in Eclipse. Some tool logging began in the middle of the *Toolsmiths* data collection (shown in light grey) and after the *Users* data collection (denoted with a *).

do so, we compared the refactoring tool usage in the *Toolsmith* data set against the tool usage in the *User* and *Everyone* data sets.

In Table 1, the "Uses" columns indicate the number of times each refactoring tool was invoked in that dataset. The "Use %" column presents the same measure as a percentage of the total number of refactorings. (The "Batched" columns are discussed in Section 3.2.) Notice that while the rank order of each tool is similar across the three data sets — RENAME, for example, always ranks first — the proportion of occurrence of the individual refactorings varies widely between *Toolsmiths* and *Users/Everyone.* In *Toolsmiths*, RENAME accounts for about 29% of all refactorings, whereas in *Users* it accounts for about 62% and in *Everyone* for about 75%. We suspect that this difference is not because *Users* and *Everyone* perform more RENAMES than *Toolsmiths*, but because *Toolsmiths* are more frequent users of the other refactoring tools.

This analysis is limited in two ways. First, each data set was gathered over a different period of time, and the tools themselves may have changed between those periods. Second, the *Users* data include both Java and non-Java RENAME and MOVE refactorings, but the *Toolsmiths* and *Everyone* data report on just Java refactorings. This may inflate actual RENAME and MOVE percentages in *Users* relative to the other two data sets.

3.2. Programmers Repeat Refactorings

We hypothesize that when programmers perform a refactoring, they typically perform several refactorings of the same kind within a short time period. For instance, a programmer may perform several EXTRACT LOCAL VARIABLES in preparation for a single EXTRACT METHOD, or may RENAME several related instance variables at once. Based on personal experience and anecdotes from programmers, we suspect that programmers often refactor several pieces of code because several related program elements may need to be refactored in order to perform a composite refactoring. In previous research, Murphy-Hill and Black built a refactoring tool that supported refactoring several program elements at once, on the assumption that this is common [8].

To determine how often programmers do in fact repeat refactorings, we used the *Toolsmiths* and the *Users* data to measure the temporal proximity of refactorings to one another. We say that refactorings of the same kind that execute within 60 seconds of each another form a *batch*. From our personal experience, we think that 60 seconds is usu-

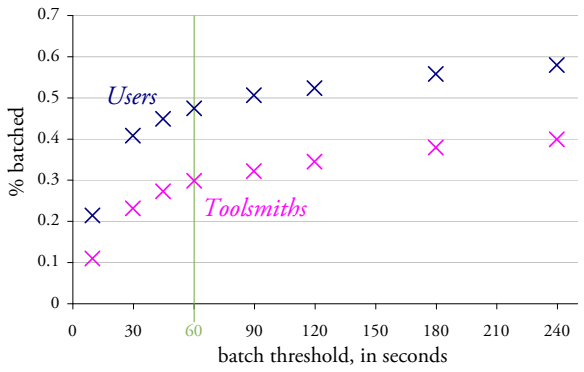

Figure 1. Percentage of refactorings that appear in batches as a function of batch threshold, in seconds. 60-seconds, the batch size used in Table 1, is drawn in green.

ally long enough to allow the programmer to complete an Eclipse wizard-based refactoring, yet short enough to exclude refactorings that are not part of the same conceptual group. Additionally, a few refactoring tools, such as PULL UP in Eclipse, can refactor multiple program elements, so a single application of such a tool is an explicit batch of related refactorings. We measured the median batch size for tools that can refactor multiple program elements in *Toolsmiths*.

In Table 1, each "Batched" column indicates the number of refactorings that appeared as part of a batch, while each "Batched %" column indicates the percentage of refactorings appearing as part of a batch. Overall, we can see that certain refactorings, such as RENAME, INTRODUCE PARAMETER, and ENCAPSULATE FIELD, are more likely to appear as part of a batch for both *Toolsmiths* and *Users*, while others, such as EXTRACT METHOD and PULL UP, are less likely to appear in a batch. In total, we see that 30% of *Toolsmiths* refactorings and 47% of *Users* refactorings appear as part of a batch.[1] For comparison, Figure 1 displays the percentage of batched refactorings for several different batch thresholds.

In *Toolsmiths*, the median batch size (for explicitly batched refactorings used with tools that can refactor multiple program elements) varied between tools. The median number of explicitly batched elements used in MOVE is 1 (n=147), PULL UP is 2.5 (n=12), PUSH DOWN is 5 (n=1), EXTRACT SUPERCLASS is 17 (n=7), and EXTRACT INTERFACE is 4.5 (n=2).

This analysis has two main limitations. First, while we wished to measure how often several related refactorings are performed in sequence, we instead used a 60-second heuristic: it may be that some related refactorings occur outside our 60-second window, and that some unrelated refactorings occur inside the window. Other metrics for detecting batches, such as burstiness, should be investigated in the future. Second, we could only ascertain how often explicit batches are used in the *Toolsmith* data set because the other data sets are not sufficiently detailed.

3.3. Programmers often don't Configure Refactoring Tools

Refactoring tools are typically of two kinds: they either force the programmer to provide configuration information, such as whether a newly created method should be `public` or `private`, or they quickly perform a refactoring without allowing any configuration. Configurable refactoring tools are more common in some environments, such as Netbeans (http://netbeans.org), whereas non-configurable tools are more common in others, such as X-develop (http://www.omnicore.com/en/xdevelop.htm). Which interface is preferable depends on how often programmers configure refactoring tools.

We hypothesize that programmers do not often configure refactoring tools. We suspect this because tweaking code manually after the refactoring may be easier than configuring the tool. In the past, we have found some limited evidence that programmers perform only a small amount of configuration of refactoring tools. When we did a small survey in September 2007 at a Portland Java User's Group meeting, 8 programmers estimated that, on average, they supply configuration information only 25% of the time.

To validate this hypothesis, we analyzed the 5 most popular refactorings performed by *Toolsmiths* to see how often programmers used various configuration options. We skipped refactorings that did not have configuration options. The results of the analysis are shown in Table 2. "Configuration Option" refers to a configuration parameter that the user can change. "Default Value" refers to the default value that the tool assigns to that option. "Change Frequency" refers to how often a user used a configuration option other than the default. The data suggest that refactoring tools are configured very little: the overall mean change frequency for these options is just under 10%. Although different configuration options are changed from defaults with varying frequencies, all configuration options that we inspected were below the average configuration percentage predicted by the Portland Java User's Group survey.

1 We suspect that the difference in percentages arises partially because the *Toolsmiths* data set counts the number of *completed* refactorings while *Users* counts the number of *initiated* refactorings. We have observed that programmers occasionally initiate a refactoring tool on some code, cancel the refactoring, and then re-initiate the same refactoring shortly thereafter [9].

Refactoring Tool	Configuration Option	Default Value	Change Frequency
Extract Local Variable	Declare the local variable as 'final'	false	5%
Extract Method	New method visibility	private	6%
	Declare thrown runtime exceptions	false	24%
	Generate method comment	false	9%
Rename Type	Update references	true	3%
	Update similarly named variables and methods	false	24%
	Update textual occurrences in comments and strings	false	15%
	Update fully qualified names in non-Java text files	true	7%
Rename Method	Update references	true	0%
	Keep original method as delegate to renamed method	false	1%
Inline Method	Delete method declaration	true	9%

Table 2. Refactoring tool configuration in Eclipse from *Toolsmiths*.

This analysis has several limitations. First, we did not have detailed-enough information in the other data sets to cross-validate our results outside *Toolsmiths*. Second, we could not count how often certain configuration options were changed, such as how often parameters are reordered when EXTRACT METHOD is performed. Third, we examined only the 5 most-common refactorings; configuration may be more frequent for less popular refactorings.

3.4. Commit Messages don't predict Refactoring

Several researchers have used messages attached to commits into a version control as indicators of refactoring activity [6, 12, 13, 15]. For example, if a programmer commits code to CVS and attaches the commit message "refactored class Foo," we might predict that the committed code contains more refactoring activity than if a programmer commits with a message that does not contain the word "refactor." However, we hypothesize that this assumption is false. We suspect this because refactoring may be an unconscious activity [2, p. 47], and because the programmer may consider it subordinate to some other activity, such as adding a feature [10].

In his thesis, Ratzinger describes the most sophisticated strategy for finding refactoring messages of which we are aware [12]: searching for the occurrence of 13 keywords, such as "move" and "rename," and excluding "needs refactoring." Using two different project histories, the author randomly drew 100 file modifications from each project and classified each as either a refactoring or as some other change. He found that his keyword technique accurately classified modifications 95.5% of the time. Based on this technique, Ratzinger and colleagues concluded that an increase in refactoring activity tends to be followed by a decrease in software defects [13].

We replicated Ratzinger's experiment for the Eclipse code base. Using the *Eclipse CVS* data, we grouped individual file revisions into global commits as previously discussed in Section 2. We also manually removed commits whose messages referred to changes to a refactoring tool (for example, "105654 [refactoring] Convert Local Variable to Field has problems with arrays"), because such changes are false positives that occur only because the project is itself a refactoring tool project. Next, using Ratzinger's 13 keywords, we automatically classified the log messages for the remaining 2788 commits. 10% of these commits matched the keywords, which compares with Ratzinger's reported 11% and 13% for two other projects [12]. Next, a third party randomly drew 20 commits from the set that matched the keywords (which we will call "Labeled") and 20 from the set that did not match ("Unlabeled"). Without knowing whether a commit was in the Labeled or Unlabeled group, two of the authors (Murphy-Hill and Parnin) manually compared each committed version of the code against the previous version, inferring how many and which refactorings were performed, and whether at least one non-refactoring change was made. Together, Murphy-Hill and Parnin compared these 40 commits over the span of about 6 hours, comparing the code using a single computer and Eclipse's standard compare tool.

The results are shown in Table 3. In the left column, the kind of **Change** is listed. Pure Whitespace means that the developer changed only whitespace or comments; No Refactoring means that the developer did not refactor but did change program behavior; Some Refactoring means that the developer both refactored and changed program behavior, and Pure Refactoring means the programmer refactored but did not change program behavior. The center column counts the number of **Labeled** commits with each kind of change, and the right column counts the number of **Unlabeled** commits. The parenthesized lists record the number

Change	Labeled	Unlabeled
Pure Whitespace	1	3
No Refactoring	8	11
Some Refactoring	5 (1,4,11,15,17)	6 (2,9,11,23,30,37)
Pure Refactoring	6 (1,1,2,3,3,5)	0
Total	20(63)	20(112)

Table 3. Refactoring between commits in *Eclipse CVS*. Plain numbers count commits in the given category; tuples contain the number of refactorings in each commit.

of refactorings found in each commit. For instance, the Table shows that in 5 commits, when a programmer mentioned a refactoring keyword in the CVS commit message, the programmer made both functional and refactoring changes. The 5 commits contained 1, 4, 11, 15, and 17 refactorings.

These results suggest that classifying CVS commits by commit message does not provide a complete picture of refactoring activity. While all 6 pure-refactoring commits were identified by commit messages that contained one of the refactoring keywords, commits labeled with a refactoring keyword contained far fewer refactorings (63, or 36% of the total) than those not so labeled (112, or 64%). Figure 2 shows the variety of refactorings in Labeled (dark blue and purple) commits and Unlabeled (light blue and pink) commits.

There are several limitations to this analysis. First, while we tried to replicate Ratzinger's experiment [12] as closely as was practicable, the original experiment was not completely specified, so we cannot say with certainty that the observed differences were not due to methodology. Likewise, observed differences may be due to differences in the projects studied. Indeed, after we completed this analysis, a personal communication with Ratzinger revealed that the original experiment included and excluded keywords specific to the projects being analyzed. Second, because the process of gathering and inspecting subsequent code revisions is labor intensive, our sample size (40 commits in total) is smaller than would otherwise be desirable. Third, the classification of a code change as a refactoring is somewhat subjective. For example, if a developer removes code known to her to never be executed, she may legitimately classify that activity as a refactoring, although to an outside observer it may appear to be the removal of a feature. We tried to be conservative, classifying changes as refactorings only when we were confident that they preserved behavior. Moreover, because the comparison was blind, any bias introduced in classification would have applied equally to both Labeled and Unlabeled commit sets.

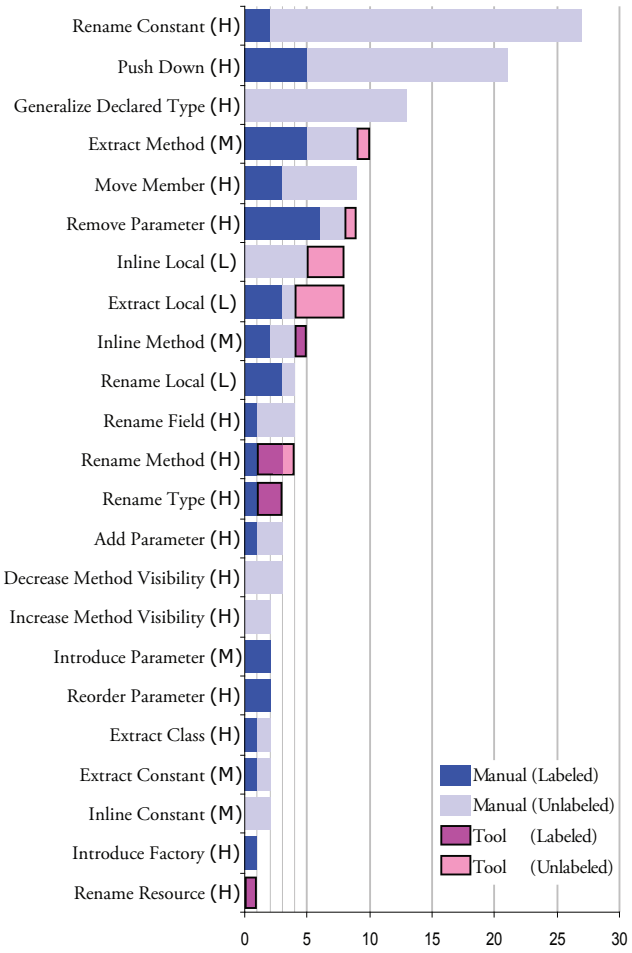

Figure 2. Refactorings over 40 sessions.

3.5. Floss Refactoring is Common

In previous work, Murphy-Hill and Black distinguished two tactics that programmers use when refactoring: floss refactoring and root-canal refactoring [10]. During floss refactoring, the programmer uses refactoring as a means to reach a specific end, such as adding a feature or fixing a bug. Thus, during floss refactoring the programmer intersperses refactoring with other kinds of program changes to keep code healthy. Root-canal refactoring, in contrast, is used for correcting deteriorated code and involves a protracted process consisting of exclusive refactoring. A survey of the literature suggested that floss refactoring is the *recommended* tactic, but provided only limited evidence that it is the more *common* tactic [10].

Why does this matter? Case studies in the literature, for example those reported by Pizka [11] and by Bourqun and Keller [1], describe root-canal refactoring. However, inferences drawn from these studies will be generally applicable

only if most refactorings are indeed root-canals.

We can estimate which refactoring tactic is used more frequently from the *Eclipse CVS* data. We first define behavioral indicators of floss and root-canal refactoring during programming sessions, which (in contrast to the intentional definitions given above) we can hope to recognize in the data. For convenience, we let a programming session be the period of time between consecutive commits to CVS by a single programmer. In a particular session, if a programmer both refactors and makes a semantic change, then we say that that the programmer is floss refactoring. If a programmer refactors during a session but does not change the semantics of the program, then we say that the programmer is root-canal refactoring. Note that a true root-canal refactoring must also last an extended period of time, or take place over several sessions. The above behavioral definitions relax this requirement, and so will tend to over-estimate the number of root canals.

Returning to Table 3, we can see that "Some Refactoring", indicative of floss refactoring, accounted for 28% of commits, while Pure Refactoring, indicative of root-canal refactoring, accounts for 15%. Normalizing for the relative frequency of commits labeled with refactoring keywords in *Eclipse CVS*, commits indicating floss refactoring would account for 30% of commits while commits indicating root-canal would account for only 3% of commits.

Also notice in Table 3 that the "Some Refactoring" (floss) row tends to show more refactorings per commit than the "Pure Refactoring" (root-canal) row. Again normalizing for labeled commits, 98% of individual refactorings would occur as part of a Some Refactoring (floss) commit, while only 2% would occur as part of a Pure Refactoring (root-canal) commit.

Pure refactoring with tools is infrequent in the *Users* data set, suggesting that very little root-canal refactoring occurred in *Users* as well. We counted the number of refactorings performed using a tool during sessions in that data. In no more than 10 out of 2671 commits did programmers use a refactoring tool without *also* manually editing their program. In other words, in less that 0.4% of commits did we observe the possibility of root-canal refactoring using only refactoring tools.

Our analysis of Table 3 is subject to the same limitations described in Section 3.4. The analysis of the *Users* data set (but not the analysis of Table 3) is also limited in that we consider only those refactorings performed using tools. Some refactorings may have been performed by hand; these would appear in the data as edits, thus possibly inflating the count of floss refactoring and reducing the count of root-canal refactoring.

	Eclipse CVS	***Toolsmiths***
Low	18%	33%
Medium	22%	27%
High	60%	40%

Table 4. Refactoring level percentages in the *Eclipse CVS* and the *Toolsmiths* data.

3.6. Many Refactorings are Medium and Low-level

Refactorings operate at a wide range of levels, from as low-level as single expressions to as high-level as whole inheritance hierarchies. Past research has often drawn conclusions based on observations of high-level refactorings. For example, several researchers have used automatic refactoring-detection tools to find refactorings in version histories, but these tools can generally detect only those refactorings that modify packages, classes, and member signatures [3, 4, 18, 19]. The tools generally do not detect sub-method level refactorings, such as EXTRACT LOCAL VARIABLE and INTRODUCE ASSERTION. We hypothesize that in practice programmers also perform many lower-level refactorings. We suspect this because lower-level refactorings will not change the program's interface and thus programmers may feel more free to perform them.

To investigate this hypothesis, we divided the refactorings that we observed in our manual inspection of *Eclipse CVS* commits into three levels — High, Medium and Low. We classified refactoring tool uses in the *Toolsmiths* data in the same way. High level refactorings are those that change the signatures of classes, methods, and fields; refactorings at this level include RENAME CLASS, MOVE STATIC FIELD, and ADD PARAMETER. Medium level refactorings are those that change the signatures of classes, methods, and fields and also significantly change blocks of code; this level includes EXTRACT METHOD, INLINE CONSTANT, and CONVERT ANONYMOUS TYPE TO NESTED TYPE. Low level refactorings are those that make changes to only blocks of code; low level refactorings include EXTRACT LOCAL VARIABLE, RENAME LOCAL VARIABLE, and ADD ASSERTION. Refactorings with tool support that were found in the *Eclipse CVS* data set are labeled as high (H), medium (M), and low (L) in Figure 2.

The results of this analysis are displayed in Table 4. For each level of refactoring, we show what percentage of refactorings from *Eclipse CVS* (normalized) and *Toolsmiths* make up that level. We see that many low and medium-level refactorings do indeed take place; as a consequence, tools that detect only high-level refactorings will miss 40 to 60 percent of refactorings.

3.7. Refactorings are Frequent

While the *concept* of refactoring is now popular, it is not entirely clear how commonly refactoring is *practiced*. In Xing and Stroulia's automated analysis of the Eclipse code base, the authors conclude that "indeed refactoring is a frequent practice" [19]. The authors make this claim largely based on observing a large number of structural changes, 70% of which are considered to be refactoring. However, this figure is based on manually excluding 75% of semantic changes — resulting in refactorings accounting for 16% of all changes. Further, their automated approach suffers from several limitations, such as the failure to detect low-level refactorings, imprecision when distinguishing signature changes from semantic changes, and the limited window of granularity offered by CVS inspection.

To validate the hypothesis that refactoring is a frequent practice, we characterize the occurrence of refactoring activity in the *Users* and *Toolsmiths* data. In order for refactoring activity to be defined as frequent, we seek to apply criteria that require refactoring to be habitual and occurring at regular intervals. For example, if refactoring activity occurs just before a software release, but not at other times, then we would not want to claim that refactoring is frequent. First, we examined the *Toolsmiths* data to determine how refactoring activity was spread throughout development. Second, we examined the *Users* data to determine how often refactoring occurred within a programming session and whether there was significant variation among the population.

In the *Toolsmiths* data, we found that refactoring activity occurred throughout the Eclipse development cycle. In 2006, an average of 30 refactorings took place each week; in 2007, there were 46 refactorings per week. Only two weeks in 2006 did not have any refactoring activity, and one of these was a winter holiday week. In 2007, refactoring occurred every week.

In the *Users* data set, we found refactoring activity distributed throughout the programming sessions. Overall, 41% of programming sessions contained refactoring activity. More interestingly, sessions that did not have refactoring activity contained an order of magnitude fewer edits than sessions with refactoring, on average. The sessions that contained refactoring also contained, on average, 71% of the total edits made by the programmer. This was consistent across the population: 22 of 31 programmers had an average greater than 72%, whereas the remaining 9 ranged from 0% to 63%. This analysis of the *Users* data suggests that when programmers must make large changes to a code base, refactoring is a common way to prepare for those changes.

Inspecting refactorings performed using a tool does not have the limitations of automated analysis; it is independent of the granularity of commits and semantic changes, and captures all levels of refactoring activity. However, it has its own limitation: the exclusion of manual refactoring. Including manual refactorings can only increase the observed frequency of refactoring. Indeed, this is likely: as we will see in Section 3.8, many refactorings are in fact performed manually.

3.8. Refactoring Tools are Underused

A programmer may perform a refactoring manually, or may choose to use an automated refactoring tool if one is available for the refactoring that she needs to perform. Ideally, a programmer will always use a refactoring tool if one is available, because automated refactorings are theoretically faster and less error-prone than manual refactorings. However, in one survey of 16 students, only 2 reported having used refactoring tools, and even then only 20% and 60% of the time [10]. In another survey of 112 agile enthusiasts, we found that the developers reported refactoring with a tool a median of 68% of the time [10]. Both of these estimates of usage are surprisingly low, but they are still only estimates. We hypothesize that programmers often do not use refactoring tools. We suspect this because existing tools may not have a sufficiently usable user-interface.

To validate this hypothesis, we correlated the refactorings that we observed by manually inspecting *Eclipse CVS* commits with the refactoring tool usages in the *Toolsmiths* data set. A refactoring found by manual inspection can be correlated with the application of a refactoring tool by looking for tool applications between commits. For example, the *Toolsmiths* data provides sufficient detail (the new variable name and location) to correlate an EXTRACT LOCAL VARIABLE with an EXTRACT LOCAL VARIABLE observed by manually inspecting adjacent commits in *Eclipse CVS*.

After analysis, we were unable to link 89% of 145 observed refactorings that had tool support to any use of a refactoring tool (also 89% when normalized). This suggests that *Toolsmiths* primarily refactor manually. An unexpected finding was that 31 refactorings that were performed with tools were not visible by comparing revisions in CVS. It appeared that most of these refactorings occurred in methods or expressions that were later removed or in newly created code that had not yet been committed to CVS. Overall, the results support the hypothesis that programmers are manually refactoring in lieu of using tools, but actual tool usage was lower than the median estimate in the professional agile developer survey. This suggests that either programmers overestimate their tool usage (perhaps refactoring is often not a conscious activity) or that expert programmers prefer to refactor manually.

This analysis suffers from two main limitations. First, it is possible that some tool usage data may be missing. If

programmers used multiple computers during development, some of which were not included in the data set, this would result in under-reporting of tool usage. Given a single commit, we could be more certain that we have a record of *all* refactoring tool uses over code in that commit if we have a record of *at least one* refactoring tool use applied to that code since the previous commit. If we apply our analysis only to those commits, then 73% of refactorings (also 73% when normalized) cannot be linked with a tool usage. Second, refactorings that occurred at an earlier time might not be committed until much later; this would inflate the count of refactorings found in CVS that we could not correlate to the use of a tool, and thus cause us to underestimate tool usage. We tried to limit this possibility by looking back several days before a commit to find uses of refactoring tools, but may not have been completely successful.

3.9. Different Refactorings are Performed with and without Tools

Some refactorings are more prone than others to being performed by hand. We have recently identified a surprising discrepancy between how programmers *want* to refactor and how they *actually* refactor using tools [10]. Programmers typically want to perform EXTRACT METHOD more often than RENAME, but programmers actually perform RENAME with tools more often than they perform EXTRACT METHOD with tools. (This can also be seen in all three groups of programmers in Table 1.) Comparing these results, we inferred that the EXTRACT METHOD tool is underused: the refactoring is instead being performed manually. However, it is unclear what other refactoring tools are underused. Moreover, there may be some refactorings that must be performed manually because no tool yet exists. We suspect that the reason that some kinds of refactoring — especially RENAME — are more often performed with tools is because these tools have simpler user interfaces.

To validate this hypothesis, we examined how the kind of refactorings differed between refactorings performed by hand and refactorings performed using a tool. We once again correlated the refactorings that we found by manually inspecting *Eclipse CVS* commits with the refactoring tool usage in the *Toolsmiths* data. In addition, when inspecting the *Eclipse CVS* commits, we identified several refactorings that currently have no tool support.

The results are shown in Figure 2. **Tool** indicates how many refactorings were performed with a tool; **Manual** indicates how many were performed without. The figure shows that manual refactorings were performed much more often for certain kinds of refactoring. For example, EXTRACT METHOD is performed 9 times manually but just once with a tool; REMOVE PARAMETER is performed 8 times manually and once with a tool. However, a few kinds of refactoring showed the opposite tendency; RENAME METHOD, for example, is most often performed with a tool. We can also see from the figure that many kinds of refactorings were performed exclusively by hand, despite having tool support.

30 refactorings did not have tool support; the most popular of these was MODIFY ENTITY PROPERTY, performed 8 times, which would allow developers to safely modify properties such as `static` or `final`. The same limitations apply as in Section 3.8.

4. Discussion

How do the results presented in Section 3 affect future refactoring research and tools?

Tool-Usage Behavior Several of our findings have reflected on the behavior of programmers using refactoring tools. For example, our finding about how toolsmiths differ from regular programmers in terms of refactoring tool usage (Section 3.1) suggests that most kinds of refactorings will not be used as frequently as the toolsmiths hoped, when compared to the frequently used RENAME refactoring. For the toolsmith, this means that improving the underused tools or their documentation (especially the tool for EXTRACT LOCAL VARIABLE) may increase tool use.

Other findings provide insight into the typical work flow involved in refactoring. Consider that refactoring tools are often used repeatedly (Section 3.2), and that programmers often do not configure refactoring tools (Section 3.3). For the toolsmith, this means that configuration-less refactoring tools, which have recently seen increasing support in Eclipse and other environments, will suit the majority of, but not all, refactoring situations. In addition, our findings about the batching of refactorings provides evidence that tools that force the programmer to repeatedly select, initiate, and configure can waste programmers' time. This was in fact one of the motivations for Murphy-Hill and Black's *refactoring cues*, a tool that allows the programmer to select several program elements for refactoring at one time [8].

Questions still remain for researchers to answer. *Why* is the RENAME refactoring tool so much more popular than other refactoring tools? Why do some refactorings tend to be batched while others do not? Moreover, our experiments should be repeated in other projects and for other refactorings to validate our findings.

Detecting Refactoring In our experiments we have investigated the assumptions underlying several commonly used refactoring-detection techniques. Unfortunately, some techniques may need refinement to address some of the concerns that we have uncovered. Our finding that commit messages in version histories are unreliable indicators of refactoring activity (Section 3.4) is at variance with an earlier finding by

Ratzinger [12]. It also casts doubt on previous research that relies on this technique [6, 13, 15]. Thus, further replication of this experiment in other contexts is needed to establish more conclusive results. Our finding that many refactorings are medium or low-level suggests that refactoring-detection techniques used by Weißgerber and Diehl [18], Dig and colleagues [4], Counsell and colleagues [3], and to a lesser extent, Xing and Stroulia [19], will not detect a significant proportion of refactorings. The effect that this has on the conclusions drawn by these authors depends on the scope of those conclusions. For example, Xing and Stroulia's conclusion that refactorings are frequent can only be bolstered when low-level refactorings are taken into consideration. On the other hand, Dig and colleagues' tool was intended to help automatically upgrade library clients, and thus has no need to find low-level refactorings. In general, researchers who wish to detect refactorings automatically should be aware of what level of refactorings their tool can detect.

Researchers can make refactoring detection techniques more comprehensive. For example, we observed that a common reason for Ratzinger's keyword-matching to misclassify changes as refactorings was that a bug-report title had been included in the commit message, and this title contained refactoring keywords. By excluding bug-report titles from the keyword search, accuracy could be increased. In general, future research can complement existing refactoring detection tools with refactoring logs from tools to increase recall of low-level refactorings.

Refactoring Practice Several of our findings bolster existing evidence about refactoring practice across a large population of programmers. Unfortunately, the findings also suggest that refactoring tools need further improvements before programmers will use them frequently. First, our finding that programmers refactor frequently (Section 3.7) confirms the same finding by Weißgerber and Diehl [18] and Xing and Stroulia [19]. For toolsmiths, this highlights the potential of refactoring tools, telling them that increased tool support for refactoring may be beneficial to programmers.

Second, our finding that floss refactoring is a more frequently practiced refactoring tactic than root-canal refactoring (Section 3.5) confirms that floss refactoring, in addition to being recommended by experts [5], is also popular among programmers. This has implications for toolsmiths, researchers, and educators. For toolsmiths, this means that refactoring tools should support flossing by allowing the programmer to switch quickly between refactoring and other development activities, which is not always possible with existing refactoring tools, such as those that force the programmer's attention away from the task at hand with modal dialog boxes [10]. For researchers, studies should focus on floss refactoring for the greatest generality. For educators, it means that when they teach refactoring to students, they should teach it throughout the course rather than as one unit during which students are taught to refactor their programs intensively.

Lastly, our findings that many refactorings are performed without the help of tools (Section 3.8) and that the kinds of refactorings performed with tools differ from the kinds performed manually (Section 3.9) confirm the results of our survey on programmers' under-use of refactoring tools [10]. Note that these findings are based on toolsmiths' refactoring tool usage, which we regard as the best case. Indeed, if even toolsmiths do not use their own refactoring tools very much, why would other programmers use them more? Toolsmiths need to explore alternative interfaces and identify common refactoring workflows, such as reminding users to EXTRACT LOCAL VARIABLE before an EXTRACT METHOD or finding a easy way to combine these refactorings: the goal should be to encourage and support programmers in taking full advantage of refactoring tools. For researchers, more study is needed about exactly *why* programmers do not use refactoring tools as much as they could.

Limitations of this Study In addition to the limitations noted in each subsection of Section 3, some characteristics of our data limit the validity of all of our analyses. First, all the data report on refactoring of Java programs in the Eclipse environment. While this is a widely-used language and environment, the results presented in this paper may not hold for other languages and environments. Second, *Users* and *Toolsmiths* may not represent programmers in general. Third, the *Users* and *Everyone* data sets may overlap the *Toolsmith* data set: both the *Users* and *Everyone* data sets were gathered from volunteers, and some of those volunteers may have been *Toolsmiths*. However, the size of the subject pools limit the impact of any overlap: fewer than 10% of the members of *Users* and 0.1% of the members of *Everyone* could be members of *Toolsmiths*.

Experimental Details Details of our methodology, including our publicly available data, the SQL queries used for correlating and summarizing that data, the tools we used for batching refactorings and grouping CVS revisions, our experimenters' notebook, and our normalization procedure, can be found at http://multiview.cs.pdx.edu/refactoring/experiments.

5. Conclusions

We have presented an analysis of four sets of data that provides new insight into how programmers refactor in practice — particularly when programmers refactor with tools and when they do not.

Some encouraging results have emerged from these data. Refactoring has been embraced by a large community of users, many of whom include refactoring as a constant companion to the development process. We observed how programmers are using refactoring tools and have found two immediate ways in which tool designers can improve refactoring tools: batching refactorings and limiting the configuration demanded by the tools.

However, there is still much work to be done. We have found evidence that suggests that researchers might have to reexamine certain assumptions about refactorings. Low- and medium-level refactorings are much more abundant, and commit messages less reliable, than previously supposed. Refactoring tools themselves are underused, particularly when we consider refactorings that have a method-level granularity or above. Future research should investigate *why* certain refactoring tools are underused and consider how this knowledge can be used to rethink these tools.

Acknowledgments. We thank Barry Anderson, Christian Bird, Tim Chevalier, Danny Dig, Markus Keller, Ralph London, Gail Murphy, Suresh Singh, and the Portland Java User's Group for their assistance, as well as the National Science Foundation for partially funding this research under CCF-0520346. Thanks to our anonymous reviewers and the participants of the Software Engineering seminar at UIUC for their excellent suggestions.

References

[1] F. Bourqun and R. K. Keller. High-impact refactoring based on architecture violations. In *CSMR '07: Proceedings of the 11th European Conference on Software Maintenance and ReEngineering*, pages 149–158, Washington, DC, USA, 2007. IEEE Computer Society.

[2] S. Counsell, Y. Hassoun, R. Johnson, K. Mannock, and E. Mendes. Trends in Java code changes: the key to identification of refactorings? In *PPPJ '03: Proceedings of the 2nd International Conference on Principles and Practice of Programming in Java*, pages 45–48, New York, NY, USA, 2003. Computer Science Press, Inc.

[3] S. Counsell, Y. Hassoun, G. Loizou, and R. Najjar. Common refactorings, a dependency graph and some code smells: an empirical study of Java OSS. In *ISESE '06: Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering*, pages 288–296, New York, NY, USA, 2006. ACM.

[4] D. Dig, C. Comertoglu, D. Marinov, and R. Johnson. Automated detection of refactorings in evolving components. In D. Thomas, editor, *ECOOP*, volume 4067 of *Lecture Notes in Computer Science*, pages 404–428. Springer, 2006.

[5] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[6] A. Hindle, D. M. German, and R. Holt. What do large commits tell us?: A taxonomical study of large commits. In *MSR '08: Proceedings of the 2008 International Workshop on Mining Software Repositories*, pages 99–108, New York, 2008. ACM.

[7] G. C. Murphy, M. Kersten, and L. Findlater. How are Java software developers using the Eclipse IDE? *IEEE Software*, 23(4):76–83, 2006.

[8] E. Murphy-Hill and A. P. Black. High velocity refactorings in Eclipse. In *Proceedings of the Eclipse Technology eXchange at OOPSLA 2007*, New York, 2007. ACM.

[9] E. Murphy-Hill and A. P. Black. Breaking the barriers to successful refactoring: Observations and tools for extract method. In *ICSE '08: Proceedings of the 30th International Conference on Software Engineering*, pages 421–430, New York, 2008. ACM.

[10] E. Murphy-Hill and A. P. Black. Refactoring tools: Fitness for purpose. *IEEE Software*, 25(5):38–44, 2008.

[11] M. Pizka. Straightening spaghetti-code with refactoring? In H. R. Arabnia and H. Reza, editors, *Software Engineering Research and Practice*, pages 846–852. CSREA Press, 2004.

[12] J. Ratzinger. *sPACE: Software Project Assessment in the Course of Evolution*. PhD thesis, Vienna University of Technology, Austria, 2007.

[13] J. Ratzinger, T. Sigmund, and H. C. Gall. On the relation of refactorings and software defect prediction. In *MSR '08: Proceedings of the 2008 International Workshop on Mining Software Repositories*, pages 35–38, New York, 2008. ACM.

[14] R. Robbes. Mining a change-based software repository. In *MSR '07: Proceedings of the Fourth International Workshop on Mining Software Repositories*, pages 15–23, Washington, DC, USA, 2007. IEEE Computer Society.

[15] K. Stroggylos and D. Spinellis. Refactoring–does it improve software quality? In *WoSQ '07: Proceedings of the 5th International Workshop on Software Quality*, pages 10–16, Washington, DC, USA, 2007. IEEE Computer Society.

[16] The Eclipse Foundation. Usage Data Collector Results, February 12, 2009. Website, http://www.eclipse.org/org/usagedata/reports/data/commands.csv.

[17] M. A. Toleman and J. Welsh. Systematic evaluation of design choices for software development tools. *Software - Concepts and Tools*, 19(3):109–121, 1998.

[18] P. Weißgerber and S. Diehl. Are refactorings less error-prone than other changes? In *MSR '06: Proceedings of the 2006 International Workshop on Mining Software Repositories*, pages 112–118, New York, 2006. ACM.

[19] Z. Xing and E. Stroulia. Refactoring practice: How it is and how it should be supported — an Eclipse case study. In *ICSM '06: Proceedings of the 22nd IEEE International Conference on Software Maintenance*, pages 458–468, Washington, DC, USA, 2006. IEEE Computer Society.

[20] T. Zimmermann and P. Weißgerber. Preprocessing CVS data for fine-grained analysis. In *MSR '04: Proceedings of the International Workshop on Mining Software Repositories*, pages 2–6, 2004.