A Comparative Study of Manual and Automated Refactorings

Stas Negara, Nicholas Chen, Mohsen Vakilian,
Ralph E. Johnson, and Danny Dig

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801, USA
{snegara2, nchen, mvakili2, rjohnson, dig}@illinois.edu

Abstract. Despite the enormous success that manual and automated refactoring has enjoyed during the last decade, we know little about the practice of refactoring. Understanding the refactoring practice is important for developers, refactoring tool builders, and researchers. Many previous approaches to study refactorings are based on comparing code snapshots, which is *imprecise*, *incomplete*, and does not allow answering research questions that involve time or compare manual and automated refactoring.
We present the first extended empirical study that considers both manual and automated refactoring. This study is enabled by our algorithm, which infers refactorings from *continuous* changes. We implemented and applied this algorithm to the code evolution data collected from 23 developers working in their natural environment for 1,520 hours. Using a corpus of 5,371 refactorings, we reveal several new facts about manual and automated refactorings. For example, more than half of the refactorings were performed manually. The popularity of automated and manual refactorings differs. More than one third of the refactorings performed by developers are clustered in time. On average, 30% of the performed refactorings do not reach the Version Control System.

1 Introduction

Refactoring [10] is an important part of software development. Development processes like eXtreme Programming [3] treat refactoring as a key practice. Refactoring has revolutionized how programmers design software: it has enabled programmers to continuously explore the design space of large codebases, while preserving the existing behavior. Modern IDEs such as Eclipse, NetBeans, IntelliJ IDEA, or Visual Studio incorporate refactoring in their top menu and often compete on the basis of refactoring support.

Several research projects [7, 17, 18, 23–25, 27, 31, 33] made strides into understanding the practice of refactoring. This is important for developers, refactoring tool builders, and researchers. Tool builders can improve the current generation of tools or design new tools to match the practice, which will help developers to perform their daily tasks more effectively. Understanding the practice also helps researchers by validating or refuting assumptions that were previously based on folklore. It can also focus the research attention on the refactorings that are popular in practice. Last, it can open new directions of research. For example, in this study we discovered that more than one third of the refactorings performed in practice are applied in a close time proximity to each other, thus forming a *cluster*. This result motivates new research into refactoring composition.

The fundamental technical problem in understanding the practice is being able to identify the refactorings that were applied by developers. There are a few approaches. One is to bring developers in the lab and watch how they refactor [24]. This has the advantage of observing all code changes, so it is precise. But this approach studies the programmers in a confined environment, for a short period of time, and thus, it is *unrepresentative*.

Another approach is to study the refactorings applied in the wild. The most common way is to analyze two Version Control System (VCS) snapshots of the code either manually [2, 7, 21, 22] or automatically [1, 4, 6, 15, 19, 29, 32]. However, the snapshot-based analysis has several disadvantages. First, it is *imprecise*. Many times refactorings overlap with editing sessions, e.g., a method is both renamed, and its method body is changed dramatically. Refactorings can also overlap with other refactorings, e.g., a method is both renamed and its arguments are reordered. The more overlap, the more noise. Our recent study [27] shows that 46% of refactored program entities are also edited or further refactored in the same commit. Second, it is *incomplete*. For example, if a method is renamed more than once, a snapshot-based analysis would only infer the last refactoring. Third, it is *impossible* to answer many empirical questions. For example, from snapshots we cannot determine how long it takes developers to refactor, and we cannot compare manual vs. automated refactorings.

Others [25, 31] have studied the practice of automated refactorings recorded by Eclipse [7, 16], but this approach does not take into account the refactorings that are applied manually. Recent studies [24, 25, 31] have shown that programmers sometimes perform a refactoring manually, even when the IDE provides an automated refactoring. Thus, this approach is *insufficient*.

We present the first empirical study that addresses these five serious limitations. We study the refactoring practice in the wild, while employing a *continuous* analysis. Such analysis tracks code changes as soon as they happen rather than inferring them from VCS snapshots. We study synergistically the practice of both manual and automated refactorings. We answer seven research questions:

RQ1: What is the proportion of manual vs. automated refactorings?
RQ2: What are the most popular automated and manual refactorings?
RQ3: How often does a developer perform manual vs. automated refactorings?
RQ4: How much time do developers spend on manual vs. automated refactorings?
RQ5: What is the size of manual vs. automated refactorings?
RQ6: How many refactorings are clustered?
RQ7: How many refactorings do not reach VCS?

Answering these empirical questions requires us to infer refactorings from continuous code changes. Recent tools [9, 14] that were developed for such inference were not designed for empirical studies. Therefore, we designed and implemented our own refactoring inference algorithm that analyzes code changes continuously. Currently, our algorithm infers ten kinds of refactorings performed either manually or automatically, but it can be easily extended to handle other refactorings as well. The inferred ten kinds of refactorings were previously reported [31] as the most popular among automated refactorings. Table 1 shows the inferred refactorings, ranging from API-level refactorings (e.g., Rename Class), to partially local (e.g., Extract Method), to completely local refactorings (e.g., Extract Local Variable). We think the inferred refactorings are representative since they are both popular and cover a wide range of common refactorings that operate on different scope levels. In the following, when we refer to refactorings we mean these ten refactoring kinds.

Table 1. Inferred refactorings. *API-level* refactorings operate on the elements of a program's API. *Partially local* refactorings operate on the elements of a method's body, but also affect the program's API. *Completely local* refactorings affect elements in the body of a single method only.

Scope	Refactoring
API-level	Encapsulate Field
	Rename Class
	Rename Field
	Rename Method
Partially local	Convert Local Variable to Field
	Extract Constant
	Extract Method
Completely local	Extract Local Variable
	Inline Local Variable
	Rename Local Variable

In our previous study [27], we continuously inferred Abstract Syntax Tree (AST) node operations, i.e., *add*, *delete*, and *update* AST node from fine-grained code edits (e.g., typing characters). In this study, we designed and implemented an algorithm that infers refactorings from these AST node operations. First, our algorithm infers high-level properties, e.g., replacing a variable reference with an expression. Then, from combination of properties it infers refactorings. For example, it infers that a local variable was inlined when it noticed that a variable declaration is deleted, and all its references are replaced with the initialization expression.

We applied our inference algorithm on the real code evolution data from 23 developers, working in their natural environment for 1,520 hours. We found that more than half of the refactorings were performed manually, and thus, the existing studies that focus on automated refactorings only might not be generalizable since they consider less than half of the total picture. We also found that the popularity of automated and manual refactorings differs. Our results present a fuller picture about the popularity of refactorings in general, which should help both researchers and tool builders to prioritize their work. Our findings provide an additional evidence that developers underuse automated refactoring tools, which raises the concern of the usability problems in these tools. We discovered that more than one third of the refactorings performed by developers are clustered. This result emphasizes the importance of researching refactoring clusters in order to identify refactoring composition patterns. Finally, we found that 30% of the performed refactorings do not reach the VCS. Thus, using VCS snapshots alone to analyze refactorings might produce misleading results.

This paper makes the following contributions:

1. We answered seven research questions about the practice of manual and automated refactoring and discovered several new facts.
2. We designed and implemented an algorithm that employs continuous change analysis to infer refactorings. Our inference algorithm and infrastructure have been successfully evaluated by the ECOOP artifact evaluation committee and found to meet expectations. Our implementation is open source and available at http://codingtracker.web.engr.illinois.edu.
3. We evaluated our algorithm on a large corpus of *real world* data.

2 Research Methodology

To answer our research questions, we employed the code evolution data that we collected as part of our previous user study [27] on 23 participants. We recruited 10 professional programmers who worked on different projects in domains such as marketing, banking, business process management, and database management. We also recruited 13 Computer Science graduate students and senior undergraduate summer interns who worked on a variety of research projects from six research labs at the University of Illinois at Urbana-Champaign.

The participants of our study have different affiliations, programming experience, and used our tool, CodingTracker [27], for different amounts of time. Consequently, the total *aggregated* data is non-homogeneous. To see whether this non-homogeneity affects our results, we divided our participants into seven groups along the three above mentioned categories. Table 2 shows the detailed statistics for each group as well as for the aggregated data. For every research question, we first present the aggregated result, and then discuss any discrepancies between the aggregated and the group results.

To collect code evolution data, we asked each participant to install the CodingTracker plug-in in his/her Eclipse IDE. During the study, CodingTracker recorded a variety of evolution data at several levels ranging from individual code edits up to the high-level events like automated refactoring invocations and interactions with Version Control System (VCS). CodingTracker employed existing infrastructure [31] to regularly upload the collected data to our centralized repository.

Table 2. Size and usage time statistics of the aggregated and individual groups.

Metric	**Group**							
	Aggregated	Affiliation		Tool usage (hours)		Programming experience (years)		
		Students	Professionals	≤ 50	> 50	< 5	$5 - 10$	> 10
Participants	23	13	10	13	10	5	11	6
Usage time, hours	1,520	1,048	471	367	1,152	269	775	458
Mean	66	81	47	28	115	54	70	76
STDEV	52	54	44	16	38	46	52	62

At the time when CodingTracker recorded the data, we did not have a refactoring inference algorithm. However, CodingTracker can accurately replay all the code editing events, thus recreating an exact replica of the evolution session that happened in reality. We replayed the coding sessions and this time, we applied our newly developed refactoring inference algorithm.

We first applied our AST node operations inference algorithm [27] on the collected raw data to represent code changes as *add*, *delete*, and *update* operations on the underlying AST. These basic AST node operations serve as input to our refactoring inference algorithm. Section 4 presents more details about our refactoring inference algorithm.

Next, we answer every research question by processing the output of the algorithm with the question-specific analyzer. Note that our analyzers for **RQ1** – **RQ5** ignore *trivial* refactorings. We consider a refactoring trivial if it affects a single line of code, e.g., renaming a variable with no uses.

3 Research Questions

RQ1: What is the proportion of manual vs. automated refactorings? Previous research on refactoring practice either predominantly focused on automated refactorings [23, 25, 31] or did not discriminate manual and automated refactorings [7,33]. Answering the question about the relative proportion of manual and automated refactorings will allow us to estimate how *representative* automated refactorings are of the total number of refactorings, and consequently, how general are the conclusions based on studying automated refactorings only.

For each of the ten refactoring kinds inferred by our algorithm, we counted how many refactorings were applied using Eclipse automated refactoring tools and how many of the inferred refactorings were applied manually. Fig. 1 shows our aggregated results. The last column represents the combined result for all the ten refactoring kinds.

Overall, our participants performed around 11% more manual than automated refactorings (2,820 vs. 2,551). Thus, research focusing on automated refactorings considers less than half of the total picture. Moreover, half of the refactoring kinds that we investigated, Convert Local Variable to Field, Extract Method,

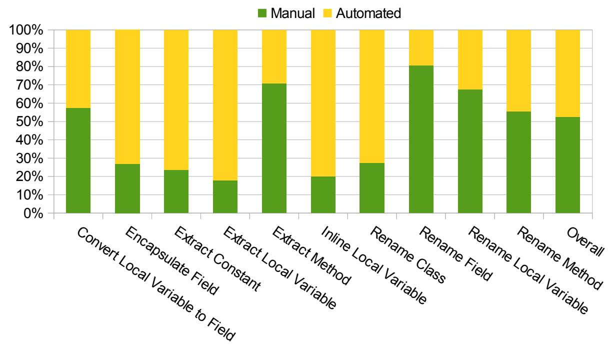


Fig. 1. Relative proportion of manual and automated refactorings.

Rename Field, Rename Local Variable, and Rename Method, are predominantly performed manually. This observation undermines the generalizability of the existing studies based on the automated execution of these popular refactorings. Also, it raises concerns for tool builders about the underuse of the automated refactoring tools, which could be a sign that these tools require a considerable improvement.

We compared the number of manual and automated refactorings performed by each group. Table 3 shows the total counts of manual and automated refactorings as well as the relative fraction of manual over automated refactorings. The results for the groups in the **Affiliation** and **Usage time** categories are consistent with the results for the aggregated data. At the same time, the programming experience of our participants has a greater impact on the ratio of the performed manual and automated refactorings. In particular, developers with less than five years of programming experience tend to perform 28% more manual than automated refactorings, while those with an average experience (5 – 10 years) perform more automated than manual refactorings. This result reflects a common intuition that novice developers are less familiar with the refactoring tools (e.g., in **RQ3** we observed that novices do not perform three kinds of automated refactorings at all), but start using them more often as their experience grows. Nevertheless, developers with more than ten years of experience perform many more manual than automated refactorings (49%). One of the reasons for this behavior could be that more experienced developers learned to perform refactorings well before the appearance of the refactoring tools. Also, experts might think that they are faster without the refactoring tool. For example, we observed that such developers mostly perform manually *Rename* refactorings, which could be accomplished quickly (but less reliably) using the *Search & Replace* command.

RQ2: What are the most popular automated and manual refactorings? Prior studies [23, 31] identified the most popular automated refactorings to better understand how developers refactor their code. We provide a more complete picture of the refactoring popularity by looking at both manual and automated refactorings. Additionally, we would like to contrast how similar or

Table 3. Manual and automated refactorings performed by each group.

Category	Group	Manual	Automated	Manual over Automated
Aggregated	All data	2820	2551	10.5%
Affiliation	Students	1645	1516	8.5%
	Professionals	1175	1035	13.5%
Usage time	$\leq$ 50 hours	485	471	3%
	$>$ 50 hours	2335	2080	12.3%
Experience	$<$ 5 years	292	228	28%
	5 – 10 years	1282	1459	-12.1%
	$>$ 10 years	1237	829	49.2%

different are popularities of automated refactorings, manual refactorings, and refactorings in general.

To measure the popularity of refactorings, we employ the same refactoring counts that we used to answer the previous research question. Fig. 2, 3, and 4 correspondingly show the popularity of automated, manual, and all refactorings in the aggregated data. The Y axis represents refactoring counts. The X axis shows refactorings ordered from the highest popularity rank at the left to the lowest rank at the right.

Our results on popularity of automated refactorings mostly corroborate previous findings [31][1]. The only exceptions are the Inline Local Variable refactoring, whose popularity has increased from the seventh to the third position, and the Encapsulate Field refactoring, whose popularity has declined from the fifth to the seventh position. Overall, our results show that the popularity of automated and manual refactorings is quite different: the top five most popular automated and manual refactorings have only three refactorings in common — Rename Local Variable, Rename Method, and Extract Local Variable, and even these refactorings have different ranks. The most important observation though is that the popularity of automated refactorings does not reflect well the popularity of refactorings in general. In particular, the top five most popular refactorings and automated refactorings share only three refactorings, out of which only one, Rename Method, has the same rank.

Having a fuller picture about the popularity of refactorings, researchers would be able to automate or infer the refactorings that are popular when considering both automated and manual refactorings. Similarly, tool builders should pay more attention to the support of the popular refactorings. Finally, novice developers might decide what refactorings to learn first depending on their relative popularity.

Refactoring popularity among different participant groups is mostly consistent with the one observed in the aggregated data. In particular, for three groups (Usage time $>$ 50 hours, Experience 5 – 10 years, and Experience $>$ 10 years), the top five most popular refactorings are the same as for the aggregated data,

[1] Note that we can not directly compare our results with the findings of Murphy et al. [23] since their data represents the related refactoring kinds as a single category (e.g., Rename, Extract, Inline, etc.).

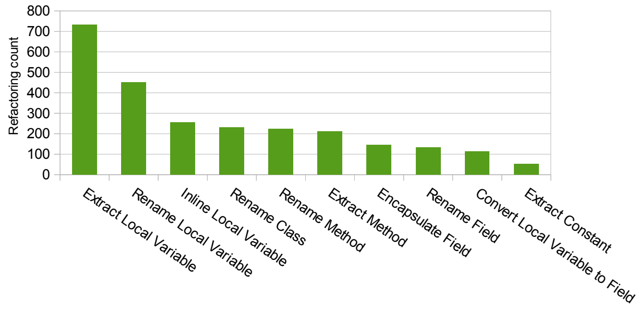


Fig. 2. Popularity of automated refactorings.

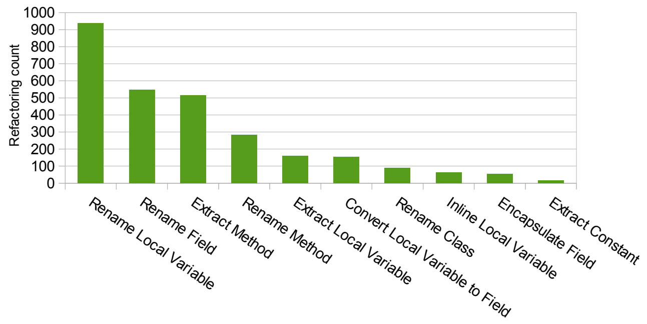


Fig. 3. Popularity of manual refactorings.

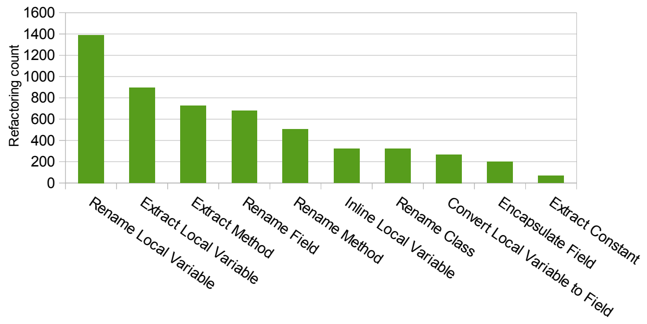


Fig. 4. Popularity of refactorings.

while for the rest of the groups, four out of five most popular refactorings are the same.

RQ3: How often does a developer perform manual vs. automated refactorings? In our previous study [31], we showed that developers may underuse automated refactoring tools for a variety of reasons, one of the most important being that developers are simply unaware of automated refactoring tools. Answering this question will help us to better understand whether developers who are aware about an automated refactoring tool use the tool rather than refactor manually.

In the following, we denote the quantity of automated tool usage as A. We compute A as a ratio of automated refactorings to the total number of refactorings of a particular kind performed by an individual participant. For each of the ten inferred refactoring kinds, we counted the number of participants for a range of values of A, from $A = 0\%$ (those who never use an automated refactoring tool) up to $A = 100\%$ (those who always use the automated refactoring tool).

Both for the aggregated data and for all groups, we observed that the fraction of participants who always perform a refactoring manually is relatively high for all the ten refactoring kinds (with a few exceptions). Also, in the aggregated data, there are no participants who apply Convert Local Variable to Field, Encapsulate Field, Extract Method, and Rename Field using the automated refactoring tools only. In groups, there are even more refactoring kinds that are never applied using automated refactoring tools only. Overall, our results provide a stronger quantitative support for the previously reported findings [25, 31] that the automated refactoring tools are underused.

To get a better insight into the practice of manual vs. automated refactoring of our participants, we defined three properties:

– **High full automation:** The number of participants who always perform the automated refactoring ($A = 100\%$) is higher than the number of participants who always perform this refactoring manually ($A = 0\%$).
– **High informed underuse:** The number of participants who are aware about the automated refactoring, but still apply it manually most of the time ($0\% < A \leq 50\%$) is higher than the number of participants who apply this refactoring automatically most of the time ($50\% < A \leq 100\%$).
– **General informed underuse:** The number of participants who apply the automated refactoring only ($A = 100\%$) is significantly lower than the number of participants who both apply the automated refactoring and refactor manually ($0\% < A < 100\%$).

Table 4 shows refactoring kinds that satisfy the above properties for each group as well as for the aggregated data. For each group, we present only the difference with the aggregated result, where "−" marks those refactoring kinds that are present in the aggregated result, but are absent in the group result, and "+" is used in the vice-versa scenario.

Our aggregated results show that only for two refactorings, Extract Constant and Rename Class, the number of participants who always perform the

Table 4. Manual vs. automated refactoring practice.

Category	Group	Property		
		High full automation	**High informed underuse**	**General informed underuse**
Aggregated		Extract Constant Rename Class	Extract Method Rename Local Variable	All
Affiliation	Students	-Extract Constant -Rename Class	+Conv. Loc. Var. to Field +Extract Constant +Rename Method	
	Professionals	+Rename Method	-Rename Local Variable	-Extract Constant -Rename Class -Rename Method
Usage time	$\leq$ 50 hours	-Extract Constant +Rename Method		-Extract Constant -Rename Class
	$>$ 50 hours	-Rename Class	+Rename Method	-Extract Constant
Experience	$<$ 5 years	-Extract Constant -Rename Class		-Conv. Loc. Var. to Field -Extract Constant -Inline Local Variable
	5 – 10 years	-Extract Constant	-Rename Local Variable +Conv. Loc. Var. to Field +Extract Constant	-Extract Constant
	$>$ 10 years	-Extract Constant	+Encapsulate Field +Rename Method	

automated refactoring is higher than the number of participants who always perform the refactoring manually. Another important observation is that for two refactoring kinds, Extract Method and Rename Local Variable, the number of participants who are aware of the automated refactoring, but still apply it manually most of the time is higher than the number of participants who apply this refactoring automatically most of the time. This shows that some automated refactoring tools are underused even when developers are aware of them and apply them from time to time. Our results for groups show that students tend to underuse more refactoring tools than professionals. Also, developers with more than five years of experience underuse more refactoring tools that they are aware of than those with less than five years of experience. At the same time, novice developers do not use three refactoring tools at all, i.e., they always perform the Convert Local Variable to Field, Extract Constant, and Inline Local Variable refactorings manually. Thus, novice developers might underuse some refactoring tools due to lack of awareness, an issue identified in a previous study [31].

The aggregated result shows that for each of the ten refactoring kinds, the number of participants who apply the automated refactoring only is significantly lower than the number of participants who both apply the automated refactoring and refactor manually. The result across all groups shows that no less than seven refactoring kinds satisfy this property. These results show that developers underuse automated refactoring tools, some more so than the others, which could be an indication of a varying degree of usability problems in these tools.

RQ4: How much time do developers spend on manual vs. automated refactorings? One of the major arguments in favor of performing a refactoring automatically is that it takes less time than performing this refactoring manually [30]. We would like to assess this time difference as well as compare the average durations of different kinds of refactorings performed manually.

To measure the duration of a manual refactoring, we consider all AST node operations that contribute to it. Our algorithm marks AST node operations that contribute to a particular inferred refactoring with a generated refactoring's ID, which allows us to track each refactoring individually. Note that a developer might intersperse a refactoring with other code changes, e.g., another refactoring, small bug fixes, etc. Therefore, to compute the duration of a manual refactoring, we cannot subtract the timestamp of the first AST node operation that contributes to it from the timestamp of the last contributing AST node operation. Instead, we compute the duration of each contributing AST node operation separately by subtracting the timestamp of the preceding AST node operation (regardless of whether it contributes to the same refactoring or not) from the timestamp of the contributing AST node operation. If the obtained duration is greater than two minutes, we discard it, since it might indicate an interruption in code editing, e.g., a developer might get distracted by a phone call or take a break. Finally, we sum up all the durations of contributing AST node operations to obtain the duration of the corresponding refactoring.

We get the durations of automated refactorings from CodingSpectator [31]. CodingSpectator measures configuration time of a refactoring performed automatically, which is the time that a developer spends in the refactoring's dialog box. Note that the measured time includes neither the time that the developer might need to check the correctness of the performed automated refactoring nor the time that it takes Eclipse to actually change the code, which could range from a couple of milliseconds to several seconds, depending on the performed refactoring kind and the underlying code.

Fig. 5 shows our aggregated results. On average, manual refactorings take longer than their automated counterparts with a high statistical significance ($p < 0.0001$, using two-sided unpaired t-test) only for Extract Local Variable, Extract Method, Inline Local Variable, and Rename Class since for the other refactoring kinds our participants rarely used the configuration dialog boxes. This observation is also statistically significant across all groups. Manual execution of the Convert Local Variable to Field refactoring takes longer than the automated one with a sufficient statistical significance ($p < 0.04$) for the aggregated data, while for most groups this observation is not statistically significant. The most time consuming, both manually and automatically, is the Extract Method refactoring, which probably could be explained by its complexity and the high amount of code changes involved. All other refactorings are performed manually on average in under $15 - 25$ seconds. Some refactorings take longer than others. A developer could take into account this difference when deciding what automated refactoring tool to learn first.

Another observation is that the Rename Field refactoring is on average the fastest manual refactoring. It takes less time than the arguably simpler Rename Local Variable refactoring. One of the possible explanations is that developers perform the Rename Field refactoring manually when it does not require many

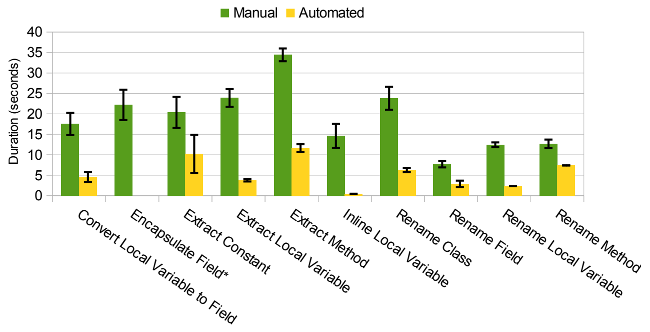


Fig. 5. Average duration of performing manual refactorings and configuring automated refactorings. The black intervals represent the standard error of the mean (SEM). The configuration time bar for the Encapsulate Field refactoring is missing since we do not have data for it.

changes, e.g., when there are few references to the renamed field, which is supported by our results for the following question.

RQ5: What is the size of manual vs. automated refactorings? In an earlier project [31], we noticed that developers mostly apply automated refactoring tools for small code changes. Therefore, we would like to compare the average size of manual and automated refactorings to better understand this behavior of developers.

To perform the comparison, we measured the size of manual and automated refactorings as the number of the affected AST nodes. For manual refactorings, we counted the number of AST node operations contributing to a particular refactoring. For automated refactorings, we counted all AST node operations that appear in between the start and the finish refactoring operations recorded by CodingTracker. Note that all operations in between the start and the finish refactoring operations represent the effects of the corresponding automated refactoring on the underlying code [27].

Fig. 6 shows our aggregated results. On average, automated refactorings affect more AST nodes than manual refactorings for four refactoring kinds, Convert Local Variable to Field, Extract Method, Rename Field, and Rename Local Variable, with a high statistical significance ($p < 0.0001$), and for three refactoring kinds, Extract Local Variable, Inline Local Variable, and Rename Method, with a sufficient statistical significance ($p < 0.03$). One of the reasons could be that developers tend to perform smaller refactorings manually since such refactorings have a smaller overhead. At the same time, this observation is not statistically significant for all the above seven refactoring kinds in every group. In particular, it is statistically significant in five out of seven groups for three refactoring kinds, Convert Local Variable to Field, Extract Method, and Rename Field, and in fewer groups for the other four kinds of refactorings.

Intuitively, one could think that developers perform small refactorings by hand and large refactorings with a tool. On the contrary, our findings show that developers perform manually even large refactorings. In particular, Extract Method is by far the largest refactoring performed both manually and automat-

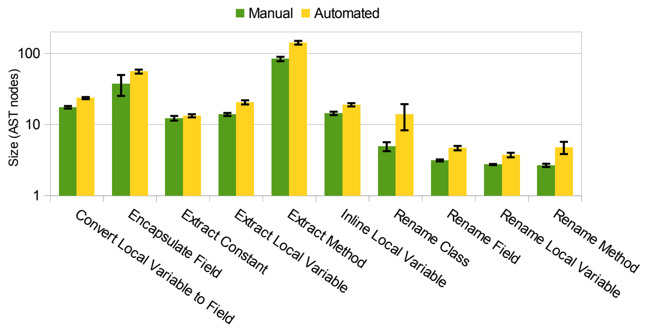


Fig. 6. Average size of manual and automated refactorings expressed as the number of the affected AST nodes. The black intervals represent the standard error of the mean (SEM). The scale of the Y axis is logarithmic.

ically – it is more than two times larger than Encapsulate Field, which is the next largest refactoring. At the same time, according to our result for **RQ3**, most of the developers predominantly perform the Extract Method refactoring manually in spite of the significant amount of the required code changes. Thus, the size of a refactoring is not a decisive factor for choosing whether to perform it manually or with a tool. This also serves as an additional indication that the developers might not be satisfied with the existing automation of the Extract Method refactoring [24].

RQ6: How many refactorings are clustered? To better understand and support refactoring activities of developers, Murphy-Hill et al. [25] identified different refactoring patterns, in particular, *root canal* and *floss* refactorings. A root canal refactoring represents a consecutive sequence of refactorings that are performed as a separate task. Floss refactorings, on the contrary, are interspersed with other coding activities of a developer. In general, grouping several refactorings in a single cluster might be a sign of a higher level refactoring pattern, and thus, it is important to know how many refactorings belong to such clusters.

To detect whether several refactorings belong to the same cluster, we compute a ratio of the number of AST node operations that are part of these refactorings to the number of AST node operations that happen in the same time window as these refactorings, but do not belong to them (such operations could happen either in between refactorings or could be interspersed with them). If this ratio is higher than a particular threshold, T, we consider that the refactorings belong to the same cluster. That is, rather than using a specific time window, we try to get as large clusters as possible, adding refactorings to a cluster as long as the ratio of refactoring to non-refactoring changes in the cluster does not fall below a particular threshold. The minimum size of a cluster is three. Note that for the clustering analysis we consider automated refactorings of all kinds and manual refactorings of the ten kinds inferred by our tool.

Fig. 7 shows the proportion of clustered and separate refactorings for aggregated data for different values of T, which we vary from 1 to 10. $T = 1$ means that the amount of non-refactoring changes does not exceed the amount of refactoring changes in the same cluster. Fig. 8 shows the average size of gaps between separate refactorings (i.e., refactorings that do not belong to any cluster) expressed as the number of AST node operations that happen in between two separate refactorings or a separate refactoring and a cluster.

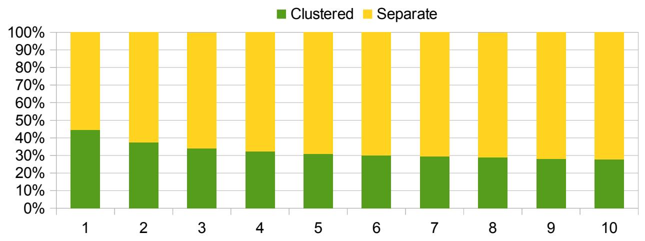


Fig. 7. Proportion of clustered and separate refactorings for different values of the threshold T.

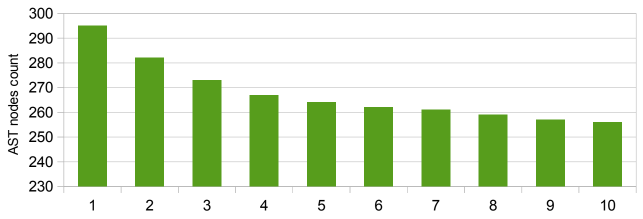


Fig. 8. The average size of gaps between separate refactorings expressed as the number of AST node operations. The X axis represents the values of the threshold T.

Our aggregated results show that for $T = 1$, 45% of the refactorings are clustered. When the threshold grows, the number of the clustered refactorings goes down, but not much — even for $T = 10$, 28% of refactorings are clustered. The average gap between floss refactorings is not very sensitive to the value of the threshold as well. Overall, developers tend to perform a significant fraction of refactorings in clusters. This observation holds for all groups except for the novice developers, where for $T = 1$ only 8% of the refactorings are clustered. One of the reasons could be that novices tend to refactor sporadically, while more experienced developers perform refactorings in chunks, probably composing them to accomplish high-level program transformations (e.g., refactor to a design pattern). Our results emphasize the importance of researching refactoring clusters in order to identify refactoring composition patterns.

RQ7: How many refactorings do not reach VCS? Software evolution researchers [6, 8, 11–13, 20, 34] use file-based Version Control Systems (VCSs), e.g., Git, SVN, CVS, as a convenient way to access the code histories of different applications. In our previous study [27], we showed that VCS snapshots provide incomplete and imprecise evolution data. In particular, we showed that 37% of code changes do not reach VCS. Since refactorings play an important role in software development, in this study, we would like to assess the amount of refactorings that never make it to VCS, and thus, are missed by any analysis based on VCS snapshots. Note that in our previous study [27] we looked at how much automated refactorings are interspersed with other code changes in the same commit in the aggregated data only, while in this study, we look at both automated and manual refactorings, we distinguish ten refactoring kinds, we distinguish different groups of participants, and we are able to count individual refactorings that are *completely* missing in VCS (rather than just being partially overlapped with some other changes).

We consider that a refactoring does not reach VCS if none of the AST node operations that are part of this refactoring reach VCS. An AST node operation does not reach VCS if there is another, later operation that affects the same node, up to the moment the file containing this node is committed to VCS. These non-reaching AST node operations and refactorings are essentially *shadowed* by other changes. For example, if a program entity is renamed twice before the code is committed to VCS, the first Rename refactoring is completely shadowed by the second one.

Fig. 9 shows the ratio of reaching and shadowed refactorings for the aggregated data. Since even a reaching refactoring might be partially shadowed, we also compute the ratio of reaching and shadowed AST node operations that are part of reaching refactorings, which is shown in Fig. 10.

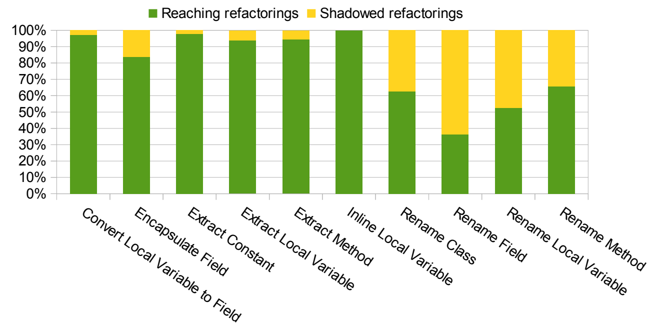


Fig. 9. Ratio of reaching and shadowed refactorings.

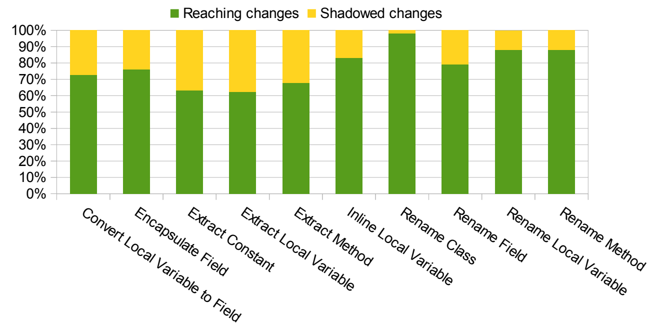


Fig. 10. Ratio of reaching and shadowed AST node operations that are part of reaching refactorings.

Our aggregated results show that for all refactoring kinds except Inline Local Variable, there is some fraction of refactorings that are shadowed. Overall, 30%

of refactorings are *completely* shadowed. The highest shadowing ratio is for the Rename refactorings. In particular, 64% of the Rename Field refactorings do not reach VCS. Thus, using VCS snapshots to analyze these refactoring kinds might significantly skew the analysis results.

Although we did not expect to see any noticeable difference between manual and automated refactorings, our results show that there are significantly more shadowed manual than automated refactorings for each refactoring kind (except Inline Local Variable, which does not have any shadowed refactorings at all). Overall, 40% of manual and only 16% of automated refactorings are shadowed. This interesting fact requires further research to understand why developers underuse automated refactorings more in code editing scenarios whose changes are unlikely to reach VCS.

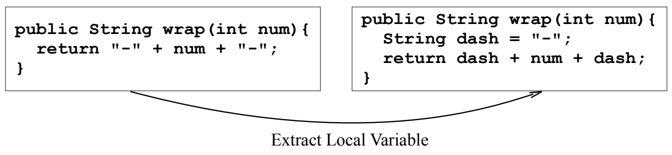
Another observation is that even refactorings that reach VCS might be hard to infer from VCS snapshots, since a noticeable fraction of AST node operations that are part of them do not reach VCS. This is particularly characteristic to the Extract refactorings, which have the highest ratio of shadowed AST node operations.

Our results for all groups are consistent with the aggregated results with a few exceptions. In particular, the percentage of completely shadowed refactorings for those participants who used our tool for less than 50 hours, is relatively small — 12%, which could be attributed to the fact that such participants did not have many opportunities to commit their code during the timespan of our study. Another observation is that for novice developers, the fraction of completely shadowed automated refactorings is significantly higher than the fraction of completely shadowed manual refactorings (38% vs. 10%). One of the reasons could be that novices experiment more with automated refactoring tools while learning them (e.g., they might perform an inappropriate automated refactoring and then undo it). Also, novice developers might be less confident in their refactoring capabilities and thus, try to see the outcome of an automated refactoring before deciding how (and whether) to refactor their code, which confirms our previous finding [31]. On the contrary, for the developers with more than ten years of programming experience, the amount of completely shadowed automated refactorings is very low — 2%, while the amount of completely shadowed manual refactorings is much higher — 39%. Thus, the most experienced developers tend to perform automated refactorings in code editing scenarios whose changes are likely to reach VCS.

4 Refactoring Inference Algorithm

4.1 Algorithm Overview

Inferring Migrated AST Nodes. Many kinds of refactorings that we would like to infer rearrange elements in the refactored program. To correctly infer such refactorings, we need to track how AST nodes migrate in the program's AST. Fig. 11 shows an example of the Extract Local Variable refactoring that results in *many-to-one* migration of the extracted AST node. Fig. 12 shows the effect of this refactoring on the underlying AST. Note that the extracted AST node, string literal `"-"`, is deleted from two places in the old AST and inserted in a single place in the new AST — as the initialization of the newly created local variable.



```
public String wrap(int num){
  return "-" + num + "-";
}
```

```
public String wrap(int num){
  String dash = "-";
  return dash + num + dash;
}
```

Extract Local Variable

Fig. 11. An example of the Extract Local Variable refactoring that results in many-to-one migration of the extracted AST node.

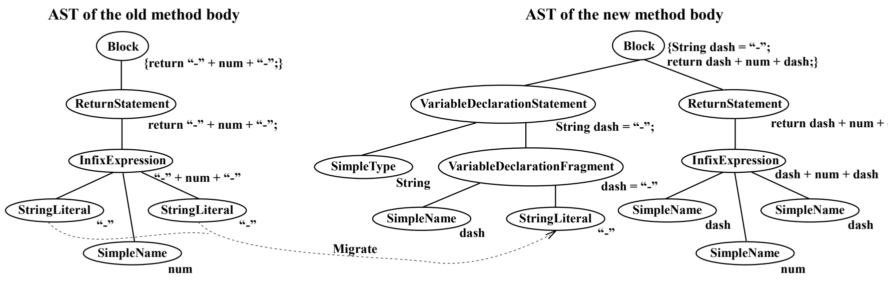
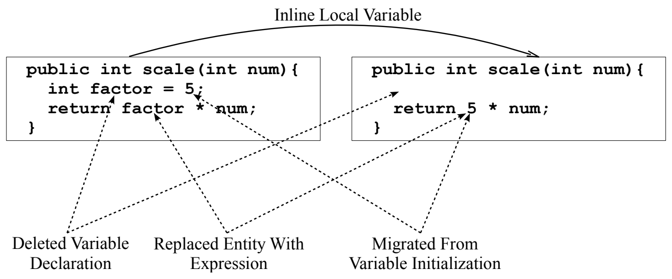


Fig. 12. The effect of the Extract Local Variable refactoring presented in Fig. 11 on the underlying AST.

Our refactoring inference algorithm infers *migrate* operations from a sequence of *basic* AST node operations: *add*, *delete*, and *update*. The algorithm assigns a unique ID to each inferred *migrate* operation and marks all basic AST node operations that make part of the inferred operation with its ID.

Inferring refactorings. Our algorithm infers ten kinds of refactorings shown in Table 1. To infer a particular kind of refactoring, our algorithm looks for *properties* that are *characteristic* to it. A refactoring property is a high-level semantic code change, e.g., addition or deletion of a variable declaration. Fig. 13 shows an example of the Inline Local Variable refactoring and its characteristic properties.

Our algorithm identifies refactoring properties directly from the basic AST node operations that represent the actions of a developer. A refactoring property is described with its *attributes*, whose values are derived from the corresponding AST node operation. Our algorithm identifies 15 attributes, e.g., `entityName`, `entityNameNodeID`, `parentID`, `migrateID`, `migratedNode`, `enclosingClassNodeID`, etc. A property may contain one or more such attributes, e.g., Migrated To Usage property has attributes `migratedNode`, `migrateID`, and `parentID`, and Deleted Entity Reference property has attributes `entityName`, `entityNameNodeID`, and `parentID`. When the algorithm checks whether a property can be part of a particular refactoring, the property's attributes are matched against attributes of all other properties that already make part of this refactoring. As a basic rule, two attributes match if either they have different names or they have the same value.



Inline Local Variable
public int scale(int num){
int factor = 5;
return factor * num;
}
public int scale(int num){
return 5 * num;
}
Deleted Variable Declaration
Replaced Entity With Expression
Migrated From Variable Initialization

Fig. 13. An example of the Inline Local Variable refactoring and its characteristic properties.

Our algorithm combines two or more closely related refactoring properties in a single refactoring *fragment*. Such fragments allow to express high level properties that could not be derived from a single AST node operation. For example, replacing a reference to an entity with an expression involves two AST node operations: *delete* entity reference and *add* expression. Consequently, the corresponding refactoring fragment, Replaced Entity With Expression, contains two properties: Migrated To Usage and Deleted Entity Reference.

The algorithm considers that a refactoring is *complete* if all its *required* characteristic properties are identified within a specific time window, which in our study is five minutes. Some characteristic properties are optional, e.g., replacing field references with getters and setters in the Encapsulate Field refactoring is optional. Also, a refactoring might include several instances of the same characteristic property. For example, an Inline Local Variable refactoring applied to a variable that is used in multiple places includes several properties of migration of the variable's initialization expression to the former usage of the variable.

Putting It All Together. Fig. 14 shows a high level overview of our refactoring inference algorithm. The algorithm takes as input the sequence of basic AST node operations marked with migrate IDs, *astNodeOperations*. The output of the algorithm is a sequence of the inferred refactorings, *inferredRefactorings*.

The refactoring inference algorithm processes each basic AST node operation from *astNodeOperations* (lines 4 – 45). First, the algorithm removes old pending complete refactorings from *pendingCompleteRefactorings* and adds them to *inferredRefactorings* (line 5). A complete refactoring is considered old if no more properties were added to it within two minutes. Also, the algorithm removes timed out pending incomplete refactorings from *pendingIncompleteRefactorings* (line 6) as well as timed out pending refactoring fragments from *pendingRefac-*

```
input: astNodeOperations // the sequence of basic AST node operations
output: inferredRefactorings
1  inferredRefactoringKinds = getAllInferredRefactoringKinds();
2  inferredRefactorings = ∅; pendingCompleteRefactorings = ∅;
3  pendingIncompleteRefactorings = ∅; pendingRefactoringFragments = ∅;
4  foreach (astNodeOperation ∈ astNodeOperations) {
5   inferredRefactorings ∪= removeOldRefactorings(pendingCompleteRefactorings);
6   removeTimedOutRefactorings(pendingIncompleteRefactorings);
7   removeTimedOutRefactoringFragments(pendingRefactoringFragments);
8   newProperties = getProperties(astNodeOperation);
9   foreach (newProperty ∈ newProperties) {
10   foreach (pendingRefactoringFragment ∈ pendingRefactoringFragments) {
11    if (accepts(pendingRefactoringFragment, newProperty) {
12     addProperty(pendingRefactoringFragment, newProperty);
13     if (isComplete(pendingRefactoringFragment) {
14      remove(pendingRefactoringFragments, pendingRefactoringFragment);
15      newProperties ∪= pendingRefactoringFragment; break;
16     }
17    }
18   }
19   if (canBePartOfRefactoringFragment(newProperty) {
20    pendingRefactoringFragments ∪= createRefactoringFragment(newProperty);
21   }
22   foreach (pendingCompleteRefactoring ∈ pendingCompleteRefactorings) {
23    if (accepts(pendingCompleteRefactoring, newProperty) {
24     addProperty(pendingCompleteRefactoring, newProperty);
25     continue foreach_line9; // the property is consumed
26    }
27   }
28   foreach (pendingIncompleteRefactoring ∈ pendingIncompleteRefactorings) {
29    if (accepts(pendingIncompleteRefactoring, newProperty) {
30     newRefactoring = clone(pendingIncompleteRefactoring);
31     addProperty(newRefactoring, newProperty);
32     if (isComplete(newRefactoring) {
33      pendingCompleteRefactorings ∪= newRefactoring;
34      continue foreach_line9; // the property is consumed
35     } else pendingIncompleteRefactorings ∪= newRefactoring;
36    }
37   }
38   foreach (inferredRefactoringKind ∈ inferredRefactoringKinds) {
39    if (isCharacteristicOf(inferredRefactoringKind, newProperty) {
40     newRefactoring = createRefactoring(inferredRefactoringKind, newProperty);
41     pendingIncompleteRefactorings ∪= newRefactoring;
42    }
43   }
44  }
45 }
46 inferredRefactorings ∪= pendingCompleteRefactorings;
```

Fig. 14. Overview of our refactoring inference algorithm.

toringFragments (line 7). An incomplete refactoring or a refactoring fragment times out if it was created more than five minutes ago.

In the following step, the algorithm generates refactoring properties specific to a particular AST node operation (line 8). The algorithm processes the generated properties one by one (lines 9 – 44). First, every new property is checked against each pending refactoring fragment (lines 10 – 18). If there is a refactoring fragment that accepts the new property and becomes complete, then this refactoring fragment itself turns into a new property to be considered by the algorithm (line 15). If the new property can be part of a new refactoring fragment, the algorithm creates the fragment and adds it to *pendingRefactoringFragments* (lines 19 – 21).

Next, the algorithm tries to add the new property to pending complete refactorings (lines 22 – 27). If the new property is added to a complete refactoring, the algorithm proceeds to the next new property (line 25). Otherwise, the algorithm checks whether this property can be added to pending incomplete refactorings (lines 28 – 37). If an incomplete refactoring accepts the property, it is added to a copy of this incomplete refactoring (lines 30 – 31). If adding the new property makes the new refactoring complete, it is added to *pendingCompleteRefactorings* (line 33) and the algorithm proceeds to the next new property (line 34). Otherwise, the new refactoring is added to *pendingIncompleteRefactorings* (line 35).

If the new property does not make any of the pending incomplete refactorings complete, the algorithm creates new refactorings of the kinds that the new property is characteristic of and adds these new refactorings to *pendingIncompleteRefactorings* (lines 38 – 43).

Finally, after processing all AST node operations, the algorithm adds to *inferredRefactorings* any of the remaining pending complete refactorings (line 46).

More details about our algorithm, including the full list of properties and their component attributes as well as composition of refactorings and refactoring fragments, can be found in our technical report [26].

4.2 Evaluation of Refactoring Inference Algorithm

Unlike the authors of the other two similar tools [9, 14], we report the accuracy of our continuous refactoring inference algorithm on *real world* data. First, we evaluated our algorithm on the automated refactorings performed by our participants, which are recorded precisely by Eclipse. We considered 2,398 automated refactorings of the nine out of the ten kinds that our algorithm infers (we disabled the inference of the automated Encapsulate Field refactoring in our experiment because the inferencer did not scale for one participant, who performed many such refactorings one after another). A challenge of any inference tool is to establish the *ground truth*, and we are the first to use such a large ground truth. Our algorithm correctly inferred 99.3% of these 2,398 refactorings. The uninferred 16 refactorings represent unlikely code editing scenarios, e.g., ten of them are the Extract Local Variable refactorings in which Eclipse re-writes huge chunks of code in a single shot.

Also, we randomly sampled 16.5 hours of code development from our corpus of 1,520 hours. Each sample is a 30-minute chunk of development activity, which includes writing code, refactoring code, running tests, committing files, etc. To establish the ground truth, the second author manually replayed each sample and recorded any refactorings (of the ten kinds that we infer) that he observed. He then compared this to the numbers reported by our inference algorithm. The first and the second authors discussed any observed discrepancies and classified them as either false positives or false negatives. Table 5 shows the sampling results for each kind of the refactoring that our algorithm infers. Overall, our inference algorithm has a precision of 0.93 and a recall of 1.

Table 5. Sampling results.

Refactoring	**True positives**	**False negatives**	**False positives**
Convert Local Variable to Field	1	0	1
Encapsulate Field	0	0	0
Extract Constant	0	0	0
Extract Local Variable	8	0	0
Extract Method	2	0	1
Inline Local Variable	2	0	0
Rename Class	3	0	0
Rename Field	5	0	0
Rename Local Variable	28	0	2
Rename Method	4	0	0
Total	53	0	4

5 Threats to Validity

5.1 Experimental Setup

We encountered difficulties in recruiting a larger group of experienced programmers due to issues such as privacy, confidentiality, and lack of trust in the reliability of research tools. However, we managed to recruit 23 participants, which we consider a sufficiently big group for our kind of study. Our dataset is not publicly available due the non-disclosure agreement with our participants.

Our dataset is non-homogeneous. In particular, our participants have different affiliations, programming experience, and used CodingTracker for a various amount of time. To address this limitation, we divided our participants in seven groups along these three categories. We answered each research question for every group as well as for the aggregated data and reported the observed insignificant discrepancies.

Our results are based on the code evolution data obtained from developers who use Eclipse for Java programming. Nevertheless, we expect our results to generalize to *similar* programming environments.

We infer only ten kinds of refactorings, which is a subset of the total number of refactorings that a developer can apply. To address this limitation to some extent, we inferred those refactoring kinds that are previously reported as being the most popular among automated refactorings [31].

5.2 Refactoring Inference Algorithm

Our refactoring inference algorithm takes as input the basic AST node operations that are inferred by another algorithm [27]. Thus, any inaccuracies in the AST node operations inference algorithm could lead to imprecisions in the refactoring inference algorithm. However, we compute the precision and recall for both these algorithms applied together, and thus, account for any inaccuracies in the input of the refactoring inference algorithm.

Although the recall of our refactoring inference algorithm is very high, the precision is noticeably lower. Hence, some of our numbers might be skewed, but we believe that the precision is high enough not to undermine our general observations.

To measure the precision and recall of the refactoring inference algorithm, we sampled around 1% of the total amount of data. Although this is a relatively small fraction of the analyzed data, the sampling was random and involved 33 distinct 30-minute intervals of code development activities, thus a manual analysis of 990 minutes of real code development.

6 Related Work

6.1 Empirical Studies of Refactoring Practice

The practice of refactoring plays a vital role in software evolution and is an important area of research. Studies by Xing and Stroulia [33], and Dig and Johnson [5] estimate that 70 – 80% of all code evolution can be expressed as refactorings.

Murphy et al. [23] were the first to study the usage of automated refactoring tools. Their study provided the first empirical ranking of the relative popularities of different automated refactorings, demonstrating that some tools are used more frequently than others. Subsequently, Murphy-Hill et al.'s [25] study on the use of automated refactoring tools provided valuable insights into the use of automated refactorings in the wild by analyzing data from multiple sources.

Due to the non-intrusive nature of CodingTracker, we were able to deploy our tool to more developers for longer periods of time, providing a more complete picture of refactoring in the wild. We inferred and recorded an *order* of magnitude more manual refactoring invocations compared to Murphy-Hill et al.'s sampling-based approach. Murphy-Hill sampled 80 commits from 12 developers for a total of 261 refactoring invocations whereas our tool recorded 1,520 hours from 23 developers for a total of 5,371 refactoring invocations.

Murphy-Hill et al.'s [25] study found that (i) refactoring tools are underused and (ii) the kinds of refactorings performed manually are different from those performed using tools. Our data (see **RQ3**) corroborates both these claims. Due to the large differences in the data sets (261 from Murphy-Hill et al. vs. 5,371 from ours), it is infeasible to meaningfully compare the raw numbers for each refactoring kind. Our work also builds upon their work by providing a more detailed breakdown of the manual and automated usage of each refactoring tool according to different participant's behavior.

Vakilian et al. [30] observed that many advanced users tend to compose several refactorings together to achieve different purposes. Our results about clustered refactorings (see **RQ6**) provide additional empirical evidence of such practices.

6.2 Automated Inference of Refactorings

Traditionally, automated refactoring inference relies on comparing two different versions of source code and describing the changes between versions of code using higher-level *characteristic properties*. A refactoring is detected based on how well it matches a set of characteristic properties. Our previous tool, *RefactoringCrawler* [6], used references of program entities (instantiations, method calls, and type imports) as its set of characteristic properties. Weißgerber and Diehl [32] used names, signature analysis, and clone detection as their set of characteristic properties. More recently, Prete et al. [28] devised a template-based approach that can infer up to 63 of the 72 refactorings cataloged by Fowler [10]. Their templates involve characteristic properties such as accesses, calls, inherited fields, etc., that model code elements in Java. Their tool, *Ref-Finder*, infers the widest variety of refactorings to date.

All these approaches rely exclusively on VCS snapshots to infer refactorings. We have shown in **RQ7** that many refactorings do not reach VCS. This compromises the accuracy of inference algorithms that rely on snapshots. To address such inadequacies, our inference algorithm leverages fine-grained edits. Similar to existing approaches, our algorithm infers refactorings by *matching* a set of characteristic properties for each refactoring. In contrast to existing approaches, our properties are precise because they are constructed directly from the AST operations that are recorded on each code edit.

In parallel with our tool, Ge et al. [14] developed *BeneFactor* and Foster et al. [9] developed *WitchDoctor*. Both these tools continuously monitor code changes to detect and complete manual refactorings in *real-time*. Though conceptually similar, our tools have different goals — we infer complete refactorings, while *BeneFactor* and *WitchDoctor* focus on inferring and completing partial refactorings in real time. Thus, their tools can afford to infer fewer kinds of refactorings and with much lower accuracy. Nonetheless, both highlight the potential of using refactoring inference algorithms based on fine-grained code changes to improve the IDE. We compare our tool with the most similar tool, *WitchDoctor*, in more detail in our technical report [26].

7 Conclusions

There are many ways to learn about the practice of refactoring, such as observing and reflecting on one's own practice, observing and interviewing other practitioners, and controlled experiments. But an important way is to analyze the changes made to a program, since programmers' beliefs about what they do can be contradicted by the evidence. Thus, it is important to be able to analyze programs and determine the kind of changes that have been made. This is traditionally done by looking at the difference between snapshots. In this paper, we have shown that VCS snapshots lose information. A continuous analysis of change lets us see that refactorings tend to be clustered, that programmers often change the name of an item several times within a short period of time and perform more manual than automated refactorings.

Our algorithm for inferring change continuously can be used for purposes other than understanding refactoring. We plan to use it as the base of a programming environment that treats changes intelligently. Continuous analysis is better at detecting refactorings than analysis of snapshots, and it ought to become the standard for detecting refactorings.

Acknowledgments. We would like to thank Nathan Hirtz for guiding us in statistical sampling techniques. We also thank Darko Marinov and students in the Software Engineering seminar at UIUC for insightful comments on earlier drafts of this paper. This work was partially supported by the National Science Foundation grants number CCF-1117960 and CCF-1213091.

References

1. Antoniol, G., Penta, M.D., Merlo, E.: An automatic approach to identify class evolution discontinuities. In: IWPSE (2004)
2. Bansiya, J.: Object-Oriented Application Frameworks: Problems and Perspectives, chap. Evaluating application framework architecture structural and functional stability (1999)
3. Beck, K., Andres, C.: Extreme Programming Explained: Embrace Change (2nd Edition). Addison-Wesley Professional (2004)
4. Demeyer, S., Ducasse, S., Nierstrasz, O.: Finding refactorings via change metrics. In: OOPSLA (2000)
5. Dig, D., Johnson, R.: The role of refactorings in api evolution. In: ICSM (2005)
6. Dig, D., Comertoglu, C., Marinov, D., Johnson, R.E.: Automated detection of refactorings in evolving components. In: ECOOP (2006)
7. Dig, D., Johnson, R.: How do APIs evolve? a story of refactoring. Journal of Soft. Maint. and Evol.: Research and Practice (2006)
8. Eick, S.G., Graves, T.L., Karr, A.F., Marron, J.S., Mockus, A.: Does code decay? assessing the evidence from change management data. IEEE TSE (2001)
9. Foster, S., Griswold, W.G., Lerner, S.: WitchDoctor: IDE Support for Real-Time Auto-Completion of Refactorings. In: ICSE (2012)
10. Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley Longman Publishing Co., Inc. (1999)

11. Gall, H., Hajek, K., Jazayeri, M.: Detection of logical coupling based on product release history. In: ICSM (1998)
12. Gall, H., Jazayeri, M., Klsch, R.R., Trausmuth, G.: Software evolution observations based on product release history. In: ICSM (1997)
13. Gall, H., Jazayeri, M., Krajewski, J.: CVS release history data for detecting logical couplings. In: IWMPSE (2003)
14. Ge, X., DuBose, Q.L., Murphy-Hill, E.: Reconciling manual and automatic refactoring. In: ICSE (2012)
15. Godfrey, M.W., Zou, L.: Using origin analysis to detect merging and splitting of source code entities. IEEE TSE (2005)
16. Henkel, J., Diwan, A.: CatchUp!: Capturing and replaying refactorings to support API evolution. In: ICSE (2005)
17. Kim, M., Cai, D., Kim, S.: An empirical investigation into the role of API-level refactorings during software evolution. In: ICSE (2011)
18. Kim, M., Zimmermann, T., Nagappan, N.: A field study of refactoring challenges and benefits. In: FSE (2012)
19. Kim, S., Pan, K., Whitehead, Jr., E.J.: When functions change their names: Automatic detection of origin relationships. In: WCRE (2005)
20. Lehman, M.M., Belady, L.A. (eds.): Program evolution: processes of software change. Academic Press Professional, Inc. (1985)
21. Mattson, M., Bosch, J.: Frameworks as components: a classification of framework evolution. In: NWPER (1998)
22. Mattson, M., Bosch, J.: Three Evaluation Methods for Object-oriented Frameworks Evolution - Application, Assessment and Comparison. Tech. rep., University of Karlskrona/Ronneby, Sweden (1999)
23. Murphy, G.C., Kersten, M., Findlater, L.: How Are Java Software Developers Using the Eclipse IDE? IEEE Software (2006)
24. Murphy-Hill, E., Black, A.P.: Breaking the barriers to successful refactoring: observations and tools for extract method. In: ICSE (2008)
25. Murphy-Hill, E., Parnin, C., Black, A.P.: How we refactor, and how we know it. IEEE TSE (2012)
26. Negara, S., Chen, N., Vakilian, M., Johnson, R.E., Dig, D.: Using Continuous Code Change Analysis to Understand the Practice of Refactoring. Tech. Rep. http://hdl.handle.net/2142/33783 (2012)
27. Negara, S., Vakilian, M., Chen, N., Johnson, R.E., Dig, D.: Is it dangerous to use version control histories to study source code evolution? In: ECOOP (2012)
28. Prete, K., Rachatasumrit, N., Sudan, N., Kim, M.: Template-based reconstruction of complex refactorings. In: ICSM (2010)
29. Rysselberghe, F.V., Demeyer, S.: Reconstruction of successful software evolution using clone detection. In: IWPSE (2003)
30. Vakilian, M., Chen, N., Moghaddam, R.Z., Negara, S., Johnson, R.E.: A compositional paradigm of automating refactorings. In: ECOOP (2013)
31. Vakilian, M., Chen, N., Negara, S., Rajkumar, B.A., Bailey, B.P., Johnson, R.E.: Use, disuse, and misuse of automated refactorings. In: ICSE (2012)
32. Weißgerber, P., Diehl, S.: Identifying refactorings from source-code changes. In: ASE (2006)
33. Xing, Z., Stroulia, E.: Refactoring practice: How it is and how it should be supported - an eclipse case study. In: ICSM (2006)
34. Xing, Z., Stroulia, E.: Analyzing the evolutionary history of the logical design of object-oriented software. TSE (2005)