Using Continuous Change Analysis to Understand the Practice of Refactoring

Stas Negara Nicholas Chen Mohsen Vakilian Ralph E. Johnson Danny Dig
University of Illinois at Urbana-Champaign
{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 to answer research questions that involve time or compare manual and automated refactoring.

We present the first empirical study that considers both manual and automated refactoring. This study is enabled by our novel algorithm, which infers refactorings from *continuous* changes. We 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 5,269 refactorings, we reveal several surprising facts about how manual and automated refactorings are different. For example, some popular automated refactorings are not representative when taking into account manual refactorings. More than one third of the refactorings performed by developers are grouped. For some refactoring kinds, up to 42% of performed refactorings do not reach the Version Control System.

1. Introduction

Refactoring [11] 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 [8], NetBeans [28], IntelliJ IDEA [19], or Visual Studio [34] incorporate refactoring in their top menu and often compete on the basis of refactoring support.

Several researchers [6, 24–27, 33, 37] made strives 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. 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, we recently discovered that more than one third of the refactorings performed in practice are applied in a group, thus motivating 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 [25]. 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.

Another approach is to study the refactorings applied in the wild. The most common way is to analyze two snapshots of the code either manually [2, 6, 22, 23] or automatically [1, 5, 7, 17, 20, 30, 35]. 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] of 24 developers working for 1,652 hours 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.

A much better approach is to study the refactoring practice in the wild, while employing a *continuous* analysis. Refactoring tools like the ones in Eclipse record all automated refactorings applied by a developer [6, 18]. Recent empirical studies about the practice of refactoring [26, 33] have used these recorded logs as the source of their analysis. But this approach does not take into account the refactorings that are applied manually. Others [25, 26, 33] have shown

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

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.

that programmers sometimes prefer to perform a refactoring manually, even when the IDE provides an automated refactoring.

Our paper is the first empirical study that uses a continuous change analysis to study the practice of both manual and automated refactorings. We answer seven research questions:

1. What is the proportion of manual vs. automated refactorings?
2. What are the most popular refactorings?
3. Does a developer prefer to use automated refactorings over manual ones?
4. How much time do developers spend on manual vs. automated refactorings?
5. What is the size of manual vs. automated refactorings?
6. How many refactorings are clustered?
7. How many refactorings do not reach VCS?

To answer these questions, we designed and implemented a novel refactoring inference algorithm that analyzes code changes continuously. Currently, our algorithm infers ten kinds of refactorings performed either manually or automatically. These were previously reported [33] 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). The inferred refactorings cover a wide range of common refactorings, and we believe that our algorithm can be easily extended to handle other refactorings as well.

Our algorithm uses several refinements to infer refactorings: from fine-grained code edits (e.g., typing characters) it infers Abstract Syntax Tree (AST) node operations, i.e., *add*, *delete*, and *update* AST node. From these, it infers high-level properties, e.g., replacing a variable reference with an expression. 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. To group properties, our algorithm uses a dynamic sliding window.

Number of participants	Programming Experience (years)
1	1 - 2
4	2 - 5
11	5 - 10
6	> 10

Table 2. Programming experience of the participants.

We applied our inference algorithm on the real code evolution data from 23 developers, working in their natural environment for 1,520 hours.

This paper makes the following contributions:

1. We designed seven questions to understand the practice of manual and automated refactoring.
2. We designed, implemented, and evaluated a novel algorithm for inferring refactorings. To measure the precision and recall of our inference algorithm, we sampled 16.5 hours of code development which is approximately 1% of the whole dataset.
3. We discovered several surprising facts about how manual refactorings differ from automated ones. For example, popular automated refactorings are not representative when taking into account manual refactorings. More than one third of the refactorings performed by developers are grouped. For some refactoring kinds, up to 42% of performed refactorings do not reach the Version Control System.

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 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. We also recruited 10 programmers who worked on open source projects in different domains, including marketing, banking, business process management, and database management. Table 2 shows the programming experience of our participants[1]. In the course of our study, we collected code evolution data for 1,520 hours of code development with a mean distribution of 66 hours per programmer and a standard deviation of 52.

[1] Note that only 22 out of 23 participants filled the survey and specified their programming experience.

To collect code evolution data, we asked each participant to install CodingTracker [27] 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 CodingSpectator's infrastructure [33] to regularly upload the collected data to our centralized repository.

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 novel 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.

3. Research Questions

3.1 What is the proportion of manual vs. automated refactorings?

Previous research on refactoring practice either predominantly focused on automated refactorings [24, 26, 33] or did not discriminate manual and automated refactorings [6, 37]. 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. Additionally, we will get a better insight about the refactoring behavior of developers.

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. Figure 1 shows our results. The last column represents the combined result for all the ten refactoring kinds.

Overall, our participants performed almost equal number of manual and automated refactorings. Thus, research focusing on automated refactorings considers just a half of the total picture. Moreover, half of the refactoring kinds that we investigated, Convert Local Variable to Field, Extract Method, Rename Field, Rename Local Variable, and Rename Method, are predominantly performed manually. This observation undermines 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.

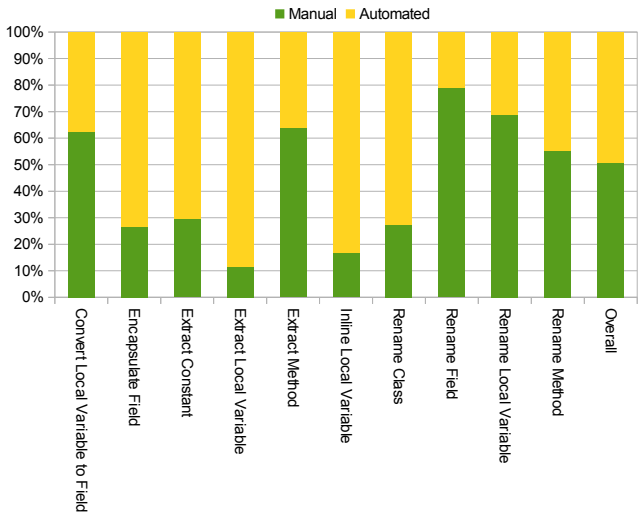

Figure 1. Relative proportion of manual and automated refactorings.

3.2 What are the most popular refactorings?

Murphy et al. [24] and Vakilian et al. [33] identified the most popular automated refactorings to better understand how developers refactor their code. We would like to get 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 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. Figures 2, 3, and 4 correspondingly show the popularity of automated, manual, and all refactorings. 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 show that the popularity of automated and manual refactorings is quite different: the top five most popular automated and manual refactorings have only two refactorings in common – Rename Local Variable and Rename Method, 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

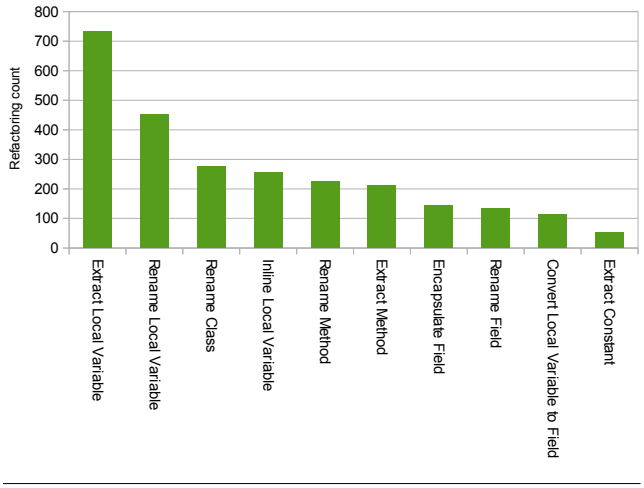

Figure 2. Popularity of automated refactorings.

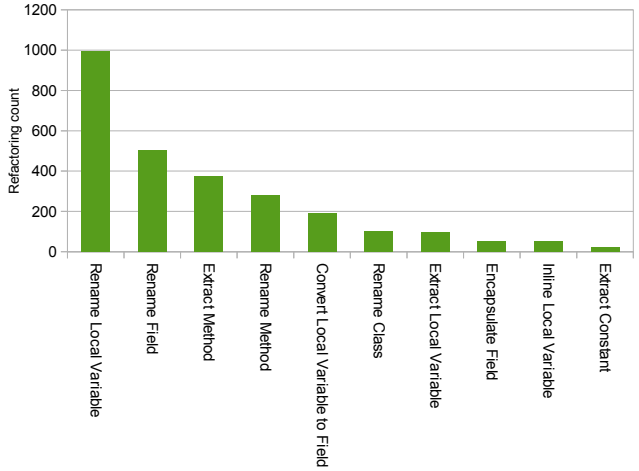

Figure 3. Popularity of manual refactorings.

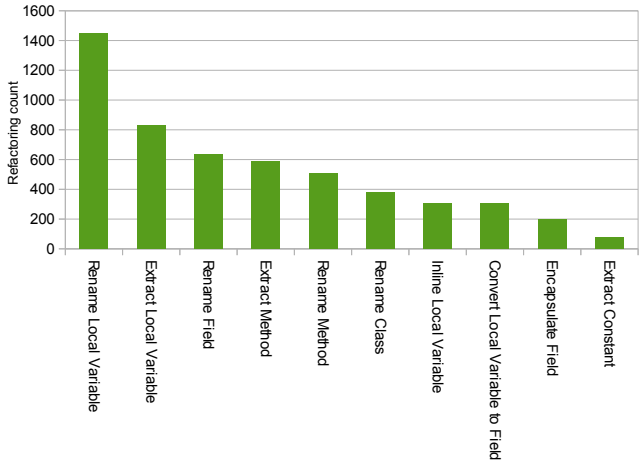

Figure 4. Popularity of refactorings.

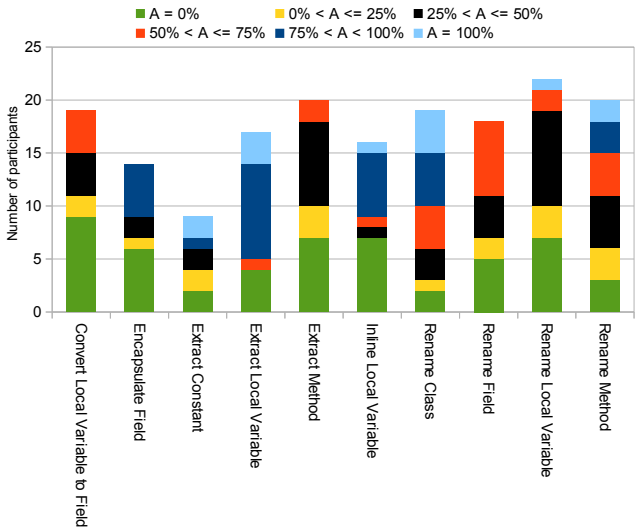

Figure 5. The degree of automated tool usage for each kind of refactoring.

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.

3.3 Does a developer prefer to use automated refactorings over manual ones?

In our previous study [33], we argued 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 prefer to 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 of the inferred refactoring kinds, we counted the number of participants who never use an automated refactoring tool ($A = 0\%$), the number of participants who predominantly refactor manually ($0\% < A <= 25\%$), the number of participants who use an automated tool quite often, but still prefer to refactor manually ($25\% < A <= 50\%$), the number of participants who prefer to refactor using an automated tool, but still often refactor manually ($50\% < A <= 75\%$), the number of participants who predominatly use an automated tool ($75\% < A < 100\%$), and the number of participants who always use the automated refactoring tool ($A = 100\%$).

Figure 5 shows our results. The Y axis represents the number of participants. Every bar shows the number of participants in each of the six automated tool usage categories, `A`, for a particular refactoring kind.

Our results show that for all refactorings, except Rename Class and Extract Constant, the number of participants who always perform the refactoring manually is higher than the number of participants who always perform the automated refactoring. Also, the fraction of participants who always perform a refactoring manually is relatively high for all the ten refactoring kinds. Overall, our results corroborate the previous findings [26, 33] that the automated refactoring tools are underused.

Another important observation is that for four refactoring kinds, Convert Local Variable to Field, Extract Constant, Extract Method, and Rename Local Variable, the number of participants who are aware about the automated refactoring, but still prefer to apply it manually ($0\% < A <= 50\%$) is higher than the number of participants who prefer to apply this refactoring automatically ($50\% < A <= 100\%$). This shows that some automated refactoring tools are underused even when developers are aware of them and apply them from time to time. Moreover, for all of the ten refactoring kinds, 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 ($0\% < A < 100\%$). In particular, there are no participants who always apply Convert Local Variable to Field, Encapsulate Field, Extract Method, and Rename Field using the automated refactoring tools. These results show that developers are reluctant to use automated refactoring tools, some more so than the others, which could be an indication of a varying degree of usability problems in these tools.

3.4 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 [32]. 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 the 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.

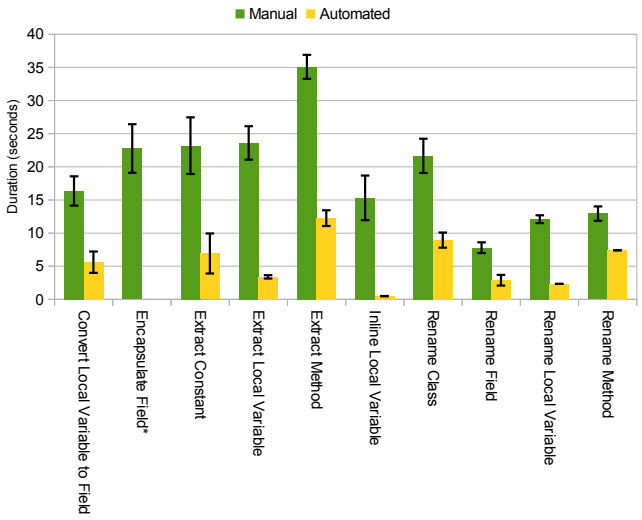

Figure 6. Average duration of performing manual refactorings and configuring automated refactorings. The black intervals represent the standard error of the mean (SEM). The configuration time bar for Encapsulate Field refactoring is missing since we do not have data for it.

We get the durations of automated refactorings from CodingSpectator [33]. 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 configuration time does not include 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.

Figure 6 shows our results. The Y axis represents the duration time in seconds. Note that the configuration time bar for Encapsulate Field refactoring is missing since we do not have data for this refactoring.

We can compare durations of automated and manual refactorings with a high statistical significance ($p < 0.0005$, using t-test ANOVA) 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. The most time consuming, both manually and automatically, is 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.

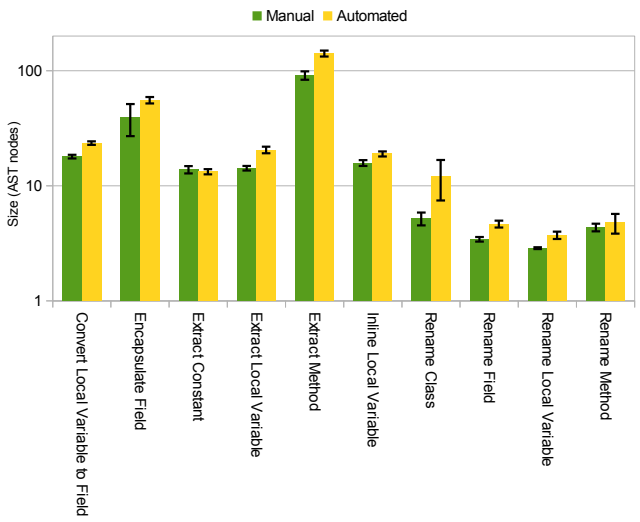

Figure 7. Average size of manual and automated refactorings expressed as the number of the affected AST nodes. The black intervals represent the standard error of the mean (SEM). The scale of the Y axis is logarithmic.

Another observation is that 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 Rename Field refactoring manually when it does not require many changes, e.g., when there are few references to the renamed field, which is supported by our results for the following question.

3.5 What is the size of manual vs. automated refactorings?

In an earlier project [33], we noticed that developers tend to 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].

Figure 7 shows our results. The logarithmic Y axis represents the number of the affected AST nodes.

Our results show that for four refactoring kinds, Convert Local Variable to Field, Extract Method, Rename Field, and Rename Local Variable, automated refactorings on average affect more AST nodes than manual refactorings with a high statistical significance ($p < 0.0005$). One of the reasons could be that developers tend to perform smaller refactorings manually since such refactorings have a smaller overhead.

Another observation is that Extract Method is by far the largest refactoring performed both manually and automatically – it is roughly three times larger than Encapsulate Field, which is the next largest refactoring. At the same time, according to Figure 5, most of the developers prefer to perform Extract Method refactoring manually in spite of the significant amount of the required code changes. This serves as an additional indication that the developers might not be happy with the existing automation of Extract Method refactoring [25].

3.6 How many refactorings are clustered?

To better understand and support refactoring activities of developers, Murphy-Hill et al. [26] 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. We try to get as large clusters as possible. 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.

Figure 8 shows the proportion of clustered and separate refactorings for different values of T, which we vary from 1 to 10. Figure 9 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.

Our results show that for $T = 1$, the majority of the refactorings are clustered. When the threshold grows, the number of the clustered refactorings goes down, but not much – even for $T = 10$, more than 35% 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 batch mode. This observation emphasizes the importance of researching refactoring clusters in order to identify refactoring composition patterns.

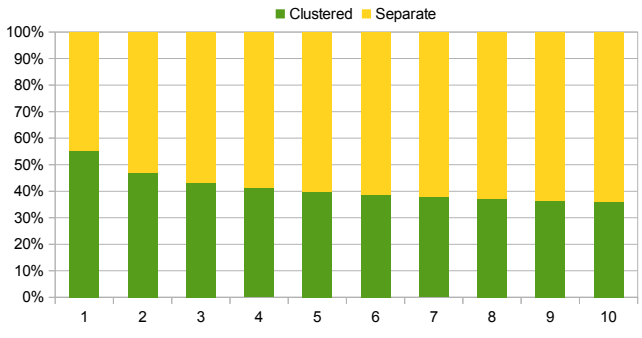

Figure 8. Proportion of clustered and separate refactorings for different values of the threshold T.

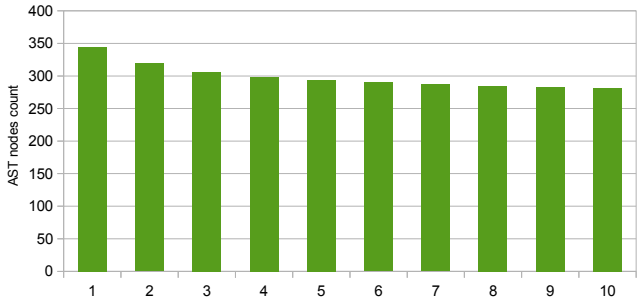

Figure 9. The average size of gaps between separate refactorings expressed as the number of AST node operations. The X axis represents the values of the threshold T.

3.7 How many refactorings do not reach VCS?

Software evolution researchers [7, 9, 12–14, 21, 36] use file-based Version Control Systems (VCSs), e.g., Git [16], SVN [31], CVS [4], 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 import 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.

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.

Figure 10 shows the ratio of reaching and shadowed refactorings. 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 Figure 11.

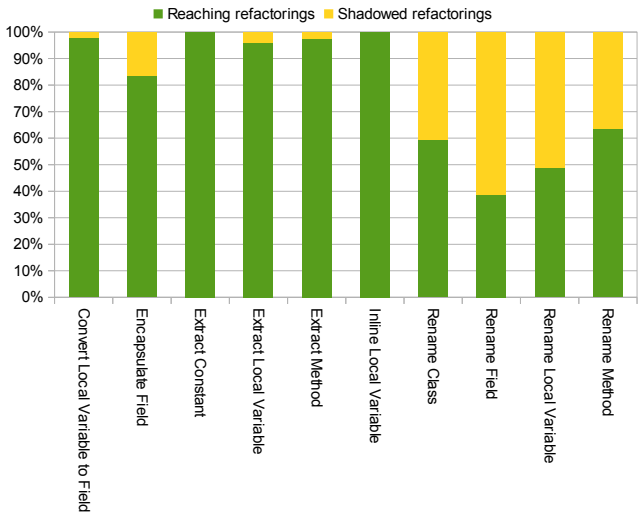

Figure 10. Ratio of reaching and shadowed refactorings.

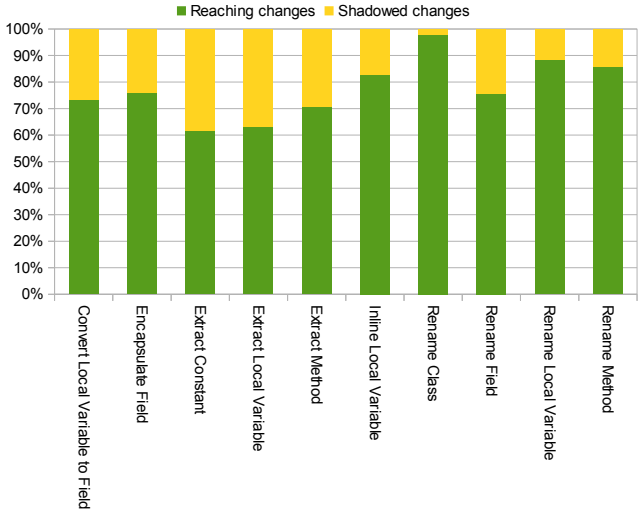

Figure 11. Ratio of reaching and shadowed AST node operations that are part of reaching refactorings.

Our results show that for all refactoring kinds except Extract Constant and Inline Local Variable, there is some fraction of refactorings that are shadowed. The highest shadowing ratio is for Rename refactorings. In particular, there are more shadowed Rename Field and Rename Local Variable refactorings than those that reach VCS. Thus, using VCS snapshots to analyze these refactoring kinds might significantly skew the analysis results.

Another observation is that even reaching refactorings 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 Extract refactorings, which have the highest ratio of shadowed AST node operations.

4. Refactoring Inference Algorithm

4.1 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. A node might migrate from a single site to another single site (i.e., this node is moved from one parent node to another parent node), for example, as a result of Inline Local Variable refactoring applied to a variable with a single usage. Such migration is *one-to-one* migration. Also, a node might migrate from a single site to multiple sites, e.g., as a result of Inline Local Variable refactoring applied to a variable with multiple usages in the code. Such migration is *one-to-many* migration. Finally, a node might migrate from multiple sites to a single site, e.g., as a result of Extract Local Variable refactoring applied to an expression that appears in multiple places in the code. Such migration is *many-to-one* migration.

Figure 12 shows an example of Extract Local Variable refactoring that results in many-to-one migration of the extracted AST node. Figure 13 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.

Our refactoring inference algorithm takes as input a sequence of *basic* AST node operations: *add*, *delete*, and *update*. The algorithm infers *migrate* operation from these basic operations. A single *migrate* operation is composed either from one *delete* operation and one or more *add* or *update* operations, or from one *add* or *update* operation and one or more *delete* operations applied on the same AST node within a specific time window. We consider that two AST nodes represent the same node if they have the same AST node type and the same content. As a time window, we employ a five minutes time interval.

The algorithm assigns a unique ID to each inferred *migrate* operation. Note that a basic AST node operation can make part of at most one *migrate* operation. The algorithm marks each basic AST node operation that makes part of a particular *migrate* operation with its ID. This allows to easily establish whether two basic AST node operations belong to the same *migrate* operation in the following stages of our refactoring inference algorithm.

4.2 Refactoring Inference Algorithm Overview

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. Figure 14 shows an example of Inline Local Variable refactoring and its characteristic properties: deletion of a variable declaration, deletion of a variable reference, and migration of the variable's initialization expression to the former usage of the variable.

Attribute name	Description
entityName	The name of a program entity
oldEntityName	The old name of a program entity
newEntityName	The new name of a program entity
migratedNode	The migrated AST node
migrateID	The ID of the migrate operation
parentID	The ID of the parent node
destinationMethodID	The ID of the destination method
sourceMethodName	The name of the source method
sourceMethodID	The ID of the source method
getterMethodName	The name of a getter method
getterMethodID	The ID of the gettern method
setterMethodName	The name of the setter method
setterMethodID	The ID of the setter method

Table 3. Attributes of refactoring properties.

Refactoring properties are identified directly from the basic AST node operations that represent the actions of a developer. A developer may change the code in any order, e.g., first delete the variable declaration and then replace its references with the initialization expression, or first replace the references and then delete the variable declaration, etc. Consequently, the order in which the properties are identified does not matter.

A refactoring property is described with its *attributes*, whose values are derived from the corresponding AST node operation. Table 3 shows 13 attributes that our algorithm employs for a variety of refactoring properties. A property may contain one or more such attributes. Table 4 presents refactoring properties and their attributes. 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. Additionally, the algorithm checks that the *disjoint* attributes have different values: `destinationMethodID` should be different from `sourceMethodID` and `getterMethodID` should be different from `setterMethodID`.

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, e.g., replacing a reference to an entity with an expression involves two AST node operations: *delete* entity reference and *add* expression. Table 5 shows the inferred refactoring fragments and their component properties.

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

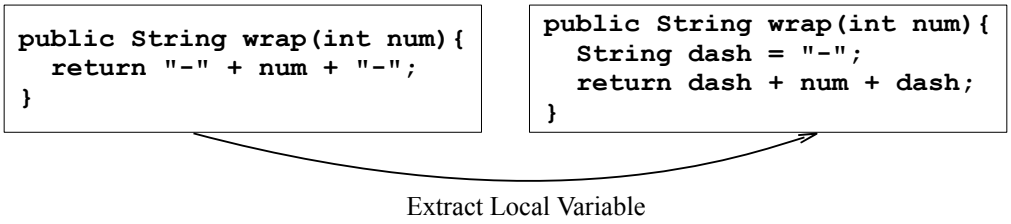

Figure 12. An example of Extract Local Variable refactoring that results in many-to-one migration of the extracted AST node.

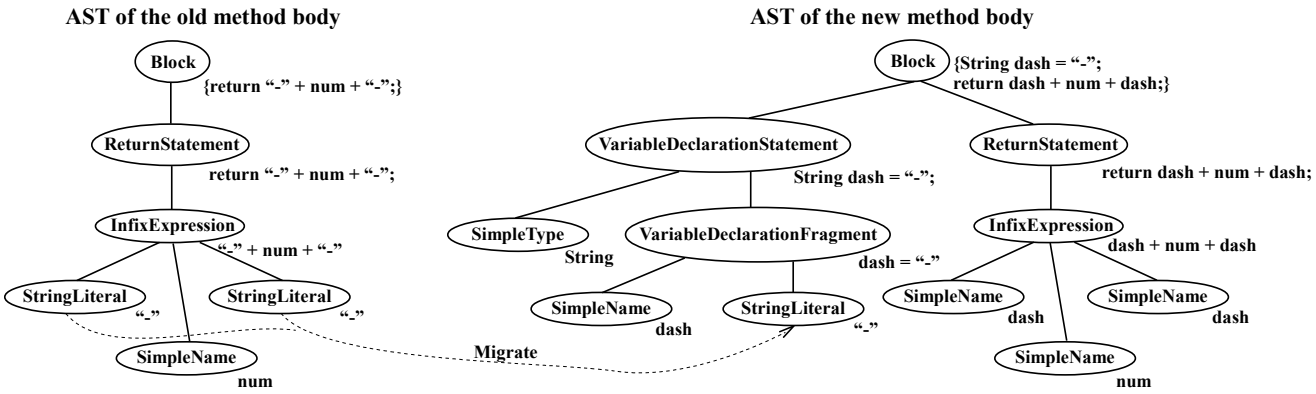

Figure 13. The effect of the Extract Local Variable refactoring presented in Figure 12 on the underlying AST.

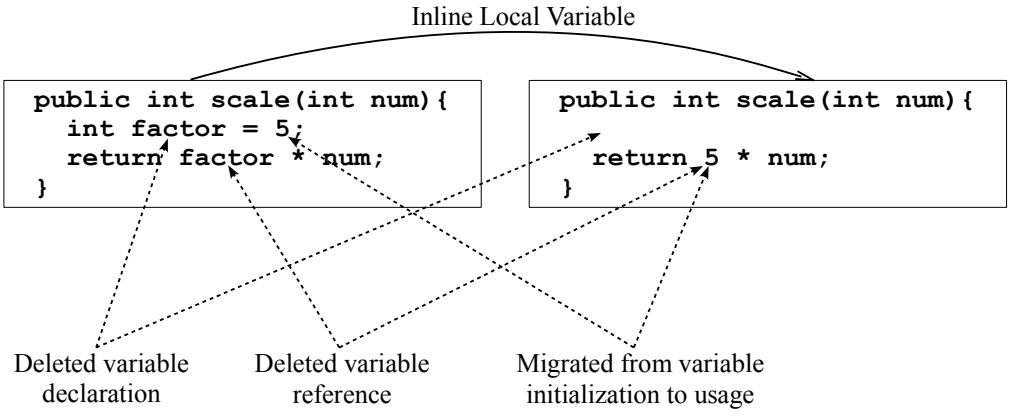

Figure 14. An example of Inline Local Variable refactoring and its characteristic properties.

Property name	Property attributes
Added Entity Reference	entityName parentID
Added Field Assignment	entityName setterMethodID
Added Field Declaration	entityName
Added Field Return	entityName getterMethodID
Added Getter Method Declaration	getterMethodName getterMethodID
Added Getter Method Invocation	getterMethodName parentID
Added Method Declaration	entityName destinationMethodID
Added Method Invocation	entityName sourceMethodName sourceMethodID
Added Setter Method Declaration	setterMethodName setterMethodID
Added Setter Method Invocation	setterMethodName parentID
Added Variable Declaration	entityName
Changed Entity Name In Usage	oldEntityName newEntityName sourceMethodName
Changed Field Name In Declaration	oldEntityName newEntityName
Changed Method Name In Declaration	oldEntityName newEntityName
Changed Type Name In Constructor	oldEntityName newEntityName
Changed Type Name In Declaration	oldEntityName newEntityName
Changed Variable Name In Declaration	oldEntityName newEntityName sourceMethodName
Deleted Entity Reference	entityName parentID
Deleted Variable Declaration	entityName
Made Field Private	entityName
Migrated From Method	sourceMethodID migrateID
Migrated From Usage	migratedNode migrateID parentID
Migrated From Variable Initialization	entityName migratedNode migrateID
Migrated To Field Initialization	entityName migratedNode migrateID
Migrated To Method	entityName destinationMethodID migrateID
Migrated To Usage	migratedNode migrateID parentID
Migrated To Variable Initialization	entityName migratedNode migrateID

Table 4. Refactoring properties.

Fragment name	Component properties
Migrated Across Methods	Migrated From Method Migrated To Method
Replaced Entity With Expression	Migrated To Usage Deleted Entity Reference
Replaced Entity With Getter	Added Getter Method Invocation Deleted Entity Reference
Replaced Entity With Setter	Added Setter Method Invocation Deleted Entity Reference
Replaced Expression With Entity	Migrated From Usage Added Entity Reference

Table 5. Refactoring fragments.

field references with getters and setters in 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. Even though it is sufficient to have a single instance of each required characteristic property to infer a refactoring, our algorithm infers a refactoring as fully as possible, incorporating all properties that belong to it. If no more properties are added to a complete refactoring within two minutes, the algorithm considers that the inference of this refactoring is finished. Table 6 presents the characteristic properties of the ten refactorings inferred by our algorithm.

4.2.1 Putting It All Together

Figure 15 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 algorithm assigns a unique ID to each inferred refactoring and marks all basic AST node operations that contribute to a refactoring with the refactoring's ID.

The refactoring inference algorithm processes each basic AST node operation from *astNodeOperations* (lines 6 – 49). First, the algorithm removes old pending complete refactorings from *pendingCompleteRefactorings* and adds them to *inferredRefactorings* (line 7). 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 8) as well as timed out pending refactoring fragments from *pendingRefactoringFragments* (line 9). An incomplete refactoring or a refactoring fragment times out if it was created more than five minutes ago, i.e., the algorithm allocates a five minutes time window for a refactoring or a refactoring fragment to become complete.

Next, the algorithm generates refactoring properties specific to a particular AST node operation (line 10). The kind

```
input: astNodeOperations // the sequence of basic AST node operations marked with migrate IDs
output: inferredRefactorings
1  inferredRefactorings = ∅;
2  inferredRefactoringKinds = getAllInferredRefactoringKinds();
3  pendingCompleteRefactorings = ∅;
4  pendingIncompleteRefactorings = ∅;
5  pendingRefactoringFragments = ∅;
6  foreach (astNodeOperation ∈ astNodeOperations) {
7    inferredRefactorings ∪= removeOldRefactorings(pendingCompleteRefactorings);
8    removeTimedOutRefactorings(pendingIncompleteRefactorings);
9    removeTimedOutRefactoringFragments(pendingRefactoringFragments);
10   newProperties = getProperties(astNodeOperation);
11   foreach (newProperty ∈ newProperties) {
12     foreach (pendingRefactoringFragment ∈ pendingRefactoringFragments) {
13       if (accepts(pendingRefactoringFragment, newProperty) {
14         addProperty(pendingRefactoringFragment, newProperty);
15         if (isComplete(pendingRefactoringFragment) {
16           remove(pendingRefactoringFragments, pendingRefactoringFragment);
17           newProperties ∪= pendingRefactoringFragment;
18           break;
19         }
20       }
21     }
22     if (canBePartOfRefactoringFragment(newProperty) {
23       pendingRefactoringFragments ∪= createRefactoringFragment(newProperty);
24     }
25     foreach (pendingCompleteRefactoring ∈ pendingCompleteRefactorings) {
26       if (accepts(pendingCompleteRefactoring, newProperty) {
27         addProperty(pendingCompleteRefactoring, newProperty);
28         continue foreach_line11; // the property is consumed
29       }
30     }
31     foreach (pendingIncompleteRefactoring ∈ pendingIncompleteRefactorings) {
32       if (accepts(pendingIncompleteRefactoring, newProperty) {
33         newRefactoring = clone(pendingIncompleteRefactoring);
34         addProperty(newRefactoring, newProperty);
35         if (isComplete(newRefactoring) {
36           pendingCompleteRefactorings ∪= newRefactoring;
37           continue foreach_line11; // the property is consumed
38         } else {
39           pendingIncompleteRefactorings ∪= newRefactoring;
40         }
41       }
42     }
43     foreach (inferredRefactoringKind ∈ inferredRefactoringKinds) {
44       if (isCharacteristicOf(inferredRefactoringKind, newProperty) {
45         newRefactoring = createRefactoring(inferredRefactoringKind, newProperty);
46         pendingIncompleteRefactorings ∪= newRefactoring;
47       }
48     }
49 }
50 inferredRefactorings ∪= pendingCompleteRefactorings;
```

Figure 15. Overview of our refactoring inference algorithm.

Refactoring	Properties/Fragments	Optional	Multiple instances
Convert Local Variable to Field	Added Field Declaration	no	no
	Deleted Variable Declaration	no	no
Encapsulate Field	Added Getter Method Declaration	no	no
	Added Setter Method Declaration	no	no
	Added Field Assignment	no	no
	Added Field Return	no	no
	Made Field Private	no	no
	Replaced Entity With Getter	yes	yes
	Replaced Entity With Setter	yes	yes
Extract Constant	Added Field Declaration	no	no
	Migrated To Field Initialization	no	no
	Replaced Expression With Entity	no	yes
Extract Local Variable	Added Variable Declaration	no	no
	Migrated To Variable Initialization	no	no
	Replaced Expression With Entity	no	yes
Extract Method	Added Method Declaration	no	no
	Added Method Invocation	no	no
	Migrated Across Methods	no	yes
Inline Local Variable	Deleted Variable Declaration	no	no
	Migrated From Variable Initialization	no	no
	Replaced Entity With Expression	no	yes
Rename Class	Changed Entity Name In Usage	yes*	yes
	Changed Type Name In Constructor	yes*	yes
	Changed Type Name In Declaration	no	no
Rename Field	Changed Entity Name In Usage	no	yes
	Changed Field Name In Declaration	no	no
Rename Local Variable	Changed Entity Name In Usage	no	yes
	Changed Variable Name In Declaration	no	no
Rename Method	Changed Entity Name In Usage	no	yes
	Changed Method Name In Declaration	no	no

Table 6. Characteristic properties of the inferred refactorings. Note that at least one of the two optional properties of the *Rename Class* refactoring, *Changed Entity Name In Usage* and *Changed Type Name In Constructor*, is required for this refactoring to be considered complete.

of the AST node operation (*add*, *delete*, or *update*), the type of the affected node (e.g., a variable declaration or reference, a method declaration, etc.), the context of the affected node (e.g., the containing method, the containing field or variable declaration, etc.), whether this operation is part of a migrate operation – all are the factors that the algorithm accounts for in order to generate one or more properties shown in Table 4.

In the following step, the algorithm processes the generated properties one by one (lines 11 – 49). First, every new property is checked against each pending refactoring fragment (lines 12 – 21). 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 17). Note that a refactoring fragment or a pending refactoring accepts a property if the property's attributes match the attributes of the properties that already make part of the fragment or the refactoring (more details on matching properties can be found in the previous subsection). If the new property can be part of a new refactoring fragment, the algorithm creates the fragment and adds it to *pendingRefactoringFragments* (lines 22 – 24).

Next, the algorithm tries to add the new property to pending complete refactorings (lines 25 – 30). If the new property is added to a complete refactoring, the algorithm proceeds to the next new property (line 28).

If there is no pending complete refactoring that accepts the new property, the algorithm checks whether this property can be added to pending incomplete refactorings (lines 31 – 42). If an incomplete refactoring accepts the property, it is added to a copy of this incomplete refactoring (lines 33 – 34). This ensures that the initial incomplete refactoring remains unchanged in *pendingIncompleteRefactorings* and thus, could be considered for future properties, if there are any. If adding the new property makes the new refactoring

complete, it is added to *pendingCompleteRefactorings* (line 36) and the algorithm proceeds to the next new property (line 37). Otherwise, the new refactoring is added to *pendingIncompleteRefactorings* (line 39).

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 43 – 48).

Finally, after processing all AST node operations, the algorithm adds to *inferredRefactorings* any of the remaining pending complete refactorings (line 50).

4.3 Evaluation of Refactoring Inference Algorithm

To evaluate our inference algorithm, we randomly sampled 16.5 hours of code development from our corpus of 1,520 hours (approximately 1%). Each sample is a 30-minute chunk of development activity, which includes writing code, refactoring code, running tests, committing files, etc. To establish *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.

The *confusion matrix* for our inference algorithm is presented below. The number of true negatives is represented as X. True negatives measure instances where a refactoring did *not* occur. Since a refactoring could occur at any time epoch (down to the last millisecond as recorded by our tool), there could be an enormous number of such true negatives. Our evaluation metrics do not depend on the number of true negatives.

		Ground truth		
		Positive	Negative	Total
Inference Algorithm	Positive	42	7	49
	Negative	1	X	$1 + X$
	Total	43	$7 + X$	

The confusion matrix allows us to calculate two standard metrics from information retrieval: *precision* and *recall*. Intuitively, precision measures how many of the inferred refactorings are indeed correct and recall measures how many of the refactorings were found by our inference algorithm. Our inference algorithm has precision of 0.86 and recall of 0.97 as calculated below.

$$\begin{aligned} Precision &= \frac{TruePositive}{TruePositive + FalsePositive} \\ &= \frac{42}{42+7} = 0.86 \\ Recall &= \frac{TruePositive}{TruePositive + FalseNegative} \\ &= \frac{42}{42+1} = 0.97 \end{aligned}$$

5. Threats to Validity

5.1 Experimental Setup

We encountered difficulties in recruiting a larger group of experienced programmers due to such issues 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.

Section 2 shows that some participants used CODINGTRACKER for longer periods of time than the others. Also, some participants might be more prolific coders or apply refactorings more often. Consequently, such participants produced a more significant impact on our results. At the same time, we think that this non-uniformity is representative of the real world.

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 small 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 the most popular among automated refactorings [33].

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 quite high, the precision is noticeably lower. As a result, some of our numbers might be skewed. Nevertheless, 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.

6. Related Work

To accurately answer questions about the practice of refactoring, we have to consider both manual and automated refactorings. Collecting information about automatic refactoring is simple and can be done through instrumenting the Eclipse refactoring infrastructure. Collecting information about manual refactorings, on the other hand, is more

complex and relies on algorithms for inferring refactorings. This section summarizes state-of-the-art work in refactoring inference and empirical research of refactoring, and contrasts our work to them.

6.1 Automatic Inference of Refactorings

Early work by Demeyer et al. [5] inferred refactorings by comparing two different versions of source code using heuristics based only on low-level software metrics (method size, class size and inheritance levels). To improve accuracy, subsequent work by other researchers described 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* [7], used references of program entities (instantiation, method calls, type imports) as its set of characteristic properties. Weißgerber and Diehl [35] used names, signature analysis, and clone detection as their set of characteristic properties. More recently, Prete et al. [29] devised a template-based approach that can infer up to 63 of the 72 refactorings cataloged by Fowler [11]. Their templates build upon 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 snapshots from VCS to infer refactorings. Thus, the accuracy of detection depends on the closeness of the two snapshots being compared. We have shown in Section 3.7 that many refactorings are shadowed and do not ever reach a commit. This compromises the accuracy of inference algorithms that rely on snapshots. Moreover, snapshot-based approaches (with the exception of *Ref-Finder*) usually concentrate only on API-level changes leaving out many of the completely or partially local refactorings that we infer (see Table 1). This paints an incomplete picture of the evolution of the code.

To address such inadequacies, our inference algorithm leverages fine-grained edits. Similar to existing approaches, our algorithm (see Table 6), infers refactorings by *matching* a set of characteristic properties for each refactoring. Our properties consists of high-level semantic changes such as adding a field, deleting a variable, etc. 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. [15] developed BeneFactor and Foster et al. [10] developed WitchDoctor. Both these tools continuously monitor code changes to detect and complete manual refactorings in *real-time*. Although conceptually similar, our tools have different goals - we infer complete refactorings, while BeneFactor and WitchDoctor try to infer and complete partial refactorings. While orthogonal to our work on studying code evolution, these projects highlight the potential of using refactoring inference algorithms based on fine-grained code changes to improve the IDE. In the following, we compare our tool with the most similar tool, WitchDoctor, in more detail.

Like our tool, WitchDoctor represents fine-grained code changes as AST node operations and uses these operations to infer refactorings. Although similar, the AST node operations and refactoring inference algorithms employed by WitchDoctor and our tool have a number of differences. In particular, our AST node operations inference algorithm [27] employs a range of heuristics for better precision, e.g., it handles Eclipse's linked edits and jumps over the unparsable state of the underlying code. WitchDoctor specifies refactorings as requirements and constraints. Our refactoring inference algorithm defines refactorings as collections of properties without explicitly specifying any constraints on them. Instead, the properties' attributes matching ensures compatibility of the properties that are part of the same refactoring (see Section 4.2). Additionally, our algorithm infers migrated AST nodes and refactoring fragments, which represent a higher level of abstraction than properties that are constructed directly from AST node operations. The authors of WitchDoctor focused on real-time performance of their tool. Since we applied our tool off-line, we were not concerned with its real-time performance, but rather assessed both precision and recall of our tool on the real world data.

6.2 Empirical Studies of Refactoring Practice

Xing and Stroulia [37] report that 70% of all changes observed in the evolution of the Eclipse code base are expressible as refactorings. Our previous study [6] of four open source frameworks and one library concluded that more than 80% of component API evolution is expressible through refactorings. These studies indicate that the practice of refactoring plays a vital role in software evolution and is an important area of research.

Our paper focuses on studying software evolution through the lens of refactoring, juxtaposing both manual and automated refactorings. Work on empirical research on the usage of automated refactoring tools was stimulated by Murphy et al.'s study [24] of 41 developers using the Java tools in Eclipse. 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 [26] comprehensive 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. As such, we were able to infer and record an *order* of magnitude more manual refactoring invocations compared to Murphy-Hill et al.'s sampling-based approach, providing a more complete picture of refactoring in the wild. To compare manual and automated refactorings, Murphy-Hill sampled 40 commits from 4 developers for a total of 175 refactoring invocations whereas our tool recorded 1,520

hours from 23 developers for a total of 5,269 refactoring invocations.

Murphy-Hill et al.'s [26] 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 Section 3.3) corroborates both these claims. We found that some refactorings are performed manually more frequently, even when the automated tools exists and the developer is aware of it. Due to the large differences in the data sets (175 from Murph-Hill et al. vs. 5,269 from ours), it is not possible to meaningfully compare the raw numbers of each refactoring kind. However, the general conclusion holds: different refactoring tools are underused at different degrees. 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. [32] observed that many advanced users tend to compose several refactorings together to achieve different purposes. Our results about clustered refactorings (see Section 3.6) provide empirical evidence of such practices. Analyzing the actual elements that are affected by each refactoring would help us better understand both how these clusters are formed and what are the implications of these clustering behaviors on software evolution.

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 always 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 sometimes snapshots hide 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, that extract method is performed more often manually than automatically.

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 based on change. Continuous analysis is better at detecting refactorings than analysis of snapshots, and it ought to become the standard for detecting refactorings.

References

[1] G. Antoniol, M. D. Penta, and E. Merlo. An automatic approach to identify class evolution discontinuities. In *IWPSE*, 2004.

[2] J. Bansiya. *Object-Oriented Application Frameworks: Problems and Perspectives*, chapter Evaluating application framework architecture structural and functional stability. 1999.

[3] K. Beck and C. Andres. *Extreme Programming Explained: Embrace Change (2nd Edition)*. Addison-Wesley Professional, 2004.

[4] CVS. CVS - Concurrent Versions System. http://cvs.nongnu.org/.

[5] S. Demeyer, S. Ducasse, and O. Nierstrasz. Finding refactorings via change metrics. In *OOPSLA*, 2000.

[6] D. Dig and R. Johnson. How do APIs evolve? a story of refactoring. *Journal of Software Maintenance and Evolution: Research and Practice*, 2006.

[7] D. Dig, C. Comertoglu, D. Marinov, and R. E. Johnson. Automated detection of refactorings in evolving components. In *ECOOP*, 2006.

[8] Eclipse. Eclipse IDE. http://www.eclipse.org/.

[9] S. G. Eick, T. L. Graves, A. F. Karr, J. S. Marron, and A. Mockus. Does code decay? assessing the evidence from change management data. *TSE*, 2001.

[10] S. Foster, W. G. Griswold, and S. Lerner. WitchDoctor: IDE Support for Real-Time Auto-Completion of Refactorings. In *To appear in ICSE*, 2012.

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

[12] H. Gall, M. Jazayeri, R. R. Klsch, and G. Trausmuth. Software evolution observations based on product release history. In *ICSM*, 1997.

[13] H. Gall, K. Hajek, and M. Jazayeri. Detection of logical coupling based on product release history. In *ICSM*, 1998.

[14] H. Gall, M. Jazayeri, and J. Krajewski. CVS release history data for detecting logical couplings. In *IWMPSE*, 2003.

[15] X. Ge, Q. L. DuBose, and E. Murphy-Hill. Reconciling manual and automatic refactoring. In *To appear in ICSE*, 2012.

[16] Git. Git - the fast version control system. http://git-scm.com/.

[17] M. W. Godfrey and L. Zou. Using origin analysis to detect merging and splitting of source code entities. *IEEE Transactions on Software Engineering*, 31(2):166–181, 2005.

[18] J. Henkel and A. Diwan. CatchUp!: Capturing and replaying refactorings to support API evolution. In *ICSE*, 2005.

[19] IntelliJIDEA. IntelliJ IDEA. http://www.jetbrains.com/idea/.

[20] S. Kim, K. Pan, and E. J. Whitehead, Jr. When functions change their names: Automatic detection of origin relationships. In *WCRE*, 2005.

[21] M. M. Lehman and L. A. Belady, editors. *Program evolution: processes of software change*. Academic Press Professional, Inc., 1985.

[22] M. Mattson and J. Bosch. Frameworks as components: a classification of framework evolution. In *NWPER*, 1998.

[23] M. Mattson and J. Bosch. Three Evaluation Methods for Object-oriented Frameworks Evolution - Application, Assessment and Comparison. Technical report, University of Karlskrona/Ronneby, Sweden, 1999.

[24] G. C. Murphy, M. Kersten, and L. Findlater. How Are Java Software Developers Using the Eclipse IDE? *IEEE Software*, 2006.

[25] E. Murphy-Hill and A. P. Black. Breaking the barriers to successful refactoring: observations and tools for extract method. In *ICSE*, 2008.

[26] E. Murphy-Hill, C. Parnin, and A. P. Black. How we refactor, and how we know it. In *ICSE*, 2009.

[27] S. Negara, M. Vakilian, N. Chen, R. E. Johnson, and D. Dig. Is it dangerous to use version control histories to study source code evolution? In *To appear in ECOOP*, 2012.

[28] NetBeans. NetBeans IDE. http://www.netbeans.org/.

[29] K. Prete, N. Rachatasumrit, N. Sudan, and M. Kim. Template-based reconstruction of complex refactorings. In *ICSM*, 2010.

[30] F. V. Rysselberghe and S. Demeyer. Reconstruction of successful software evolution using clone detection. In *IWPSE*, 2003.

[31] SVN. Apache Subversion centralized version control. http://subversion.apache.org/.

[32] M. Vakilian, N. Chen, S. Negara, R. Z. Moghaddam, and R. E. Johnson. Composite refactoring. In *Under submission to OOPSLA*, 2012.

[33] M. Vakilian, N. Chen, S. Negara, B. A. Rajkumar, B. P. Bailey, and R. E. Johnson. Use, disuse, and misuse of automated refactorings. In *To appear in ICSE*, 2012.

[34] VisualStudio. Visual Studio. http://www.microsoft.com/visualstudio/.

[35] P. Weißgerber and S. Diehl. Identifying refactorings from source-code changes. In *ASE*, 2006.

[36] Z. Xing and E. Stroulia. Analyzing the evolutionary history of the logical design of object-oriented software. *TSE*, 2005.

[37] Z. Xing and E. Stroulia. Refactoring practice: How it is and how it should be supported - an eclipse case study. In *ICSM*, 2006.