Circumventing Refactoring Masking using Fine-Grained Change Recording

Quinten David Soetens, Javier Pérez and Serge Demeyer
University of Antwerp
Antwerpen, Belgium
{quinten.soetens; javier.perez; serge.demeyer}@uantwerp.be

Andy Zaidman
Delft University of Technology
Delft, The Netherlands
a.e.zaidman@tudelft.nl

ABSTRACT

Today, refactoring reconstruction techniques are snapshot-based: they compare two revisions from a source code management system and calculate the shortest path of edit operations to go from the one to the other. An inherent risk with snapshot-based approaches is that a refactoring may be concealed by later edit operations acting on the same source code entity, a phenomenon we call *refactoring masking*. In this paper, we performed an experiment to find out at which point refactoring masking occurs and confirmed that a snapshot-based technique misses refactorings when several edit operations are performed on the same source code entity. We present a way of reconstructing refactorings using fine grained changes that are recorded live from an integrated development environment and demonstrate on two cases —PMD and Cruisecontrol— that our approach is more accurate in a significant number of situations than the state-of-the-art snapshot-based technique RefFinder.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement

Keywords

Refactoring Reconstruction; Refactoring Masking; Fine Grained Changes; Software Evolution

1. INTRODUCTION

Refactoring is widely recognised as a crucial technique applied when evolving object-oriented software systems. The key idea is to redistribute program entities and responsibilities in order to prepare the software for future extensions. If applied well, refactoring improves the design of software, makes software easier to understand, helps to find bugs, and helps to program faster [10]. As such, refactoring has received widespread attention within both academic and industrial circles, and is mentioned as a recommended practice in the software engineering body of knowledge [1].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

IWPSE'15, August 30, 2015, Bergamo, Italy
© 2015 ACM. 978-1-4503-3816-5/15/08...$15.00
http://dx.doi.org/10.1145/2804360.2804362

Given this widespread attention, several researchers set out to reconstruct refactorings as they occurred in the evolution of software projects. Initially, this was mainly an act of scientific curiosity (*i.e.*, [3, 21, 37, 39, 27, 14]), however later on actual applications emerged. Weißgerber *et al.* for instance used this as a means for studying the impact of refactorings on defects [11, 38]. Dig *et al.* prototyped a capture-playback tool capable of replaying refactorings when migrating systems dependent on a refactored API [13, 7, 8]. Obviously, several authors tried to correlate the impact of refactorings on the maintainability of a software project [32, 16, 22, 36].

In the meantime, several field studies and surveys indicated that if refactoring is applied in practice, it is mainly interwoven with normal software development [17, 15]. A side effect of this interweaving is that a commit in a source code management system tends to consist of more than just a single refactoring [2]. Indeed Negara *et al.* reported that 46% of refactored program entities are also edited in the same commit [19]. Consequently, state of the art refactoring reconstruction techniques miss a significant portion of the actual refactorings, because they infer refactorings by comparing two revisions of a system and making educated guesses about the precise edit operations applied in between. At that point, it is virtually impossible for such snapshot-based refactoring reconstruction tools to correctly deduce refactorings since these may be concealed by other changes. Negara *et al.* found that on average 30% of refactoring operations do not reach the version control system [18]. We call this the *"refactoring masking"* phenomenon and investigate the nature of the problem in a first Research Question.

RQ 1 – Refactoring Masking. *Under which conditions does a snapshot-based approach fail to reconstruct refactorings?*

To address this first research question we followed the refactoring script of a small yet representative program (the LAN Simulation [6]) and committed individual atomic changes to separate revisions in a source code repository. We then ran RefFinder [14] to compare all possible combinations of revisions to investigate under which conditions RefFinder fails to reconstruct the refactorings. We found that some refactorings indeed conceal others as they act on the same source code entities. Combinations of ExtractMethod and MoveMethod are particularly vulnerable.

A solution to this problem might be to use the actual changes, as performed in an integrated development environment. Assuming that an integrated development environment provides logging facilities for all editing operations (*i.e.*, like done in Spyware [25], Syde [12], Cheops [9], OperationRecorder [20] and ChEOPSJ [33]), we might query this stream of changes to distinguish refactorings from ordinary program edits. We performed a proof by construction via a tool prototype named ChEOPSJ, which sits in the background of Eclipse and records the changes made to a software system while a developer is programming. We compared this tool prototype against the state of the art snapshot-based approach as explained by the second research question.

RQ 2 – Comparison. *Do fine-grained changes allow us to reconstruct refactorings where snapshot-based approaches fail?*

We compare the change-based approach (exemplified by ChEOPSJ [33]) against a snapshot-based approach (exemplified by RefFinder [14]) on two open source cases – PMD and Cruisecontrol. We locate instances of the refactoring masking phenomenon and show that the change-based approach is indeed more accurate in reconstructing refactorings in those cases. Moreover, we argue that this improved accuracy is relevant by estimating the number of edit operations acting on the same source code entities within 5 minutes after an ExtractMethod refactoring.

We structured the remainder of this paper as follows. Section 2 introduces the state of the art, including a description of the ChEOPSJ tool prototype. Next we have two sections 3, and 4, which each address one of the research questions with their own experimental setup and results. The final two sections 5 and 6 wrap up the paper with a discussion of the limitations of – and threats to – the validity of our research and summarise our major findings in the conclusions.

2. STATE OF THE ART

2.1 Snapshot-Based Reconstruction

In this paper, we use the term *"refactoring reconstruction"* to refer to any software reengineering technique used to infer refactorings that were performed in the history of a software system. The current state of the art in this consists of analyses of snapshots maintained in a source code repository. Most approaches use some kind of code similarity measure to identify possible refactoring candidates. Dig *et al.* as well as Weißgerber *et al.* used a combination of a signature-based analysis and shingles (a form of hashing) [7, 8, 11, 38]. Van Rysselberghe *et al.* use clone detection on two versions to look for a decrease in the number of clones. Since many refactorings are aimed at the elimination of duplicated code [37], this would suggest that a refactoring was performed. There exist two approaches that do not rely on code similarity. Demeyer *et al.* developed a set of heuristics to identify refactorings using decreasing code entities [3]. Xing and Stroulia search for refactorings at the design level using their UMLDiff algorithm, which is capable of detecting some basic structural changes to the system [39] [27].

RefFinder, an Eclipse plugin by Kim *et al.* is to date the most comprehensive refactoring reconstruction tool as it supports 63 different types of refactorings [14]. They use the technique proposed by Prete *et al.*, which is stronger than all previous techniques because they not only detect primitive refactorings (which all previous techniques do to some extent)

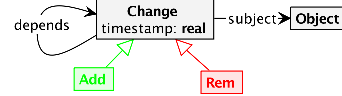

Figure 1: The types of changes implemented.

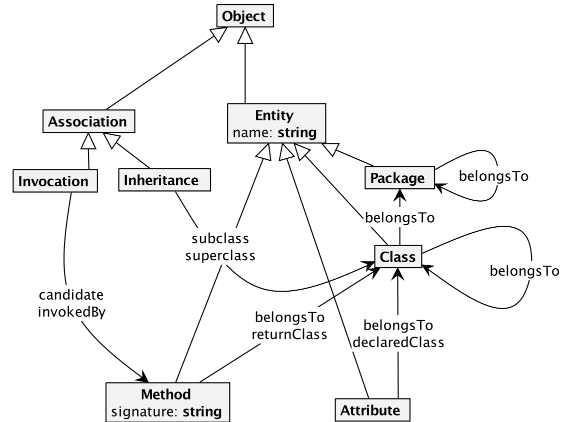

Figure 2: The types of source code entities implemented. (note: For the sake of readability, when multiple relationships exist between entities, a single arrow is drawn.)

but also "complex refactorings" (*i.e.*, refactorings which are combinations of primitive refactorings). To do this they rely on a fact base with a strong query engine (Tyruba logic) [21]. They describe the structural constraints before and after applying a refactoring in terms of template logic queries. RefFinder takes two versions of a system as input from the Eclipse workspace and recovers changes as logic facts about the systems' syntactic structure using LSDiff. These are then stored in a factbase, which can be queried to identify program differences that match the constraints of each refactoring type under focus.

We opted to use RefFinder in our experiments, because it is a representative of the state-of-the art in snapshot-based refactoring reconstruction techniques and because the list of refactorings it is able to detect is currently the most comprehensive.

2.2 Change-Based Reconstruction

An alternative to the snapshot-based approach is to use the actual edit operations as performed in an integrated development environment. In such an approach a tool silently records the activities of the programmers while they are working, and registers all the changes as performed. For instance, if the programmer modifies a method, the recorder instantiates change objects for each of the statements that were added, changed or removed. This approach was used by Robbes and Lanza in Spyware [25]; by Hattori and Lanza in Syde [12]; by Omori and Maruyama in OperationRecorder and OperationReplayer [20]; and by Ebraert *et al.* in ChEOPS [9].

We extended the later approach with our tool prototype ChEOPSJ[1], which is a Java version of the same model [34, 33]. It operates in the Eclipse background and silently records the

[1]ChEOPSJ: Change and Evolution Oriented Programming Support for Java (`http://win.ua.ac.be/~qsoeten/other/cheopsj/`)

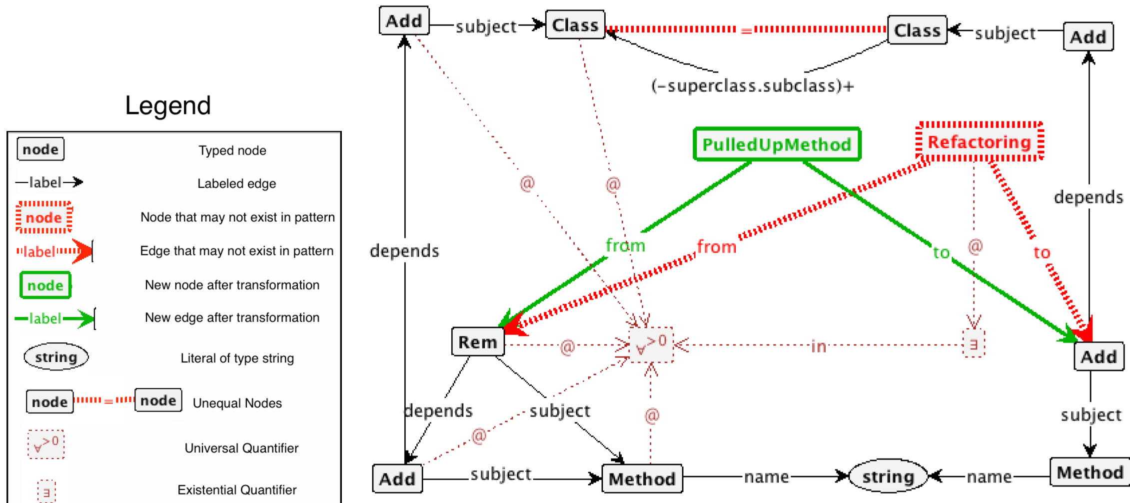

Figure 3: The graph transformation rule to reconstruct the PullUpMethod refactoring

changes that are made to the source code while the developer is programming.

In our tool we implemented two kinds of *Atomic Changes*: `Add` and `Remove` (see Figure 1). These act upon a *Subject* that represents an actual entity in the source code. For these subjects we implemented a subset of the FAMIX model [5] (see Figure 2). We chose the FAMIX model as this model captures most object oriented programming languages. It defines entities representing packages, classes, methods and attributes, as well as more fine grained entities such as method invocations, variable accesses and inheritance relationships.

In our change model the changes are interconnected through dependencies. These dependencies between change objects (See Figure 1) are determined by the relationships between the entities in the subset of the FAMIX model we are using (See Figure 2). Hence, the dependencies between change objects are defined as follows: A change c_1 is said to depend on another change c_2 if the application of c_1 without c_2 would violate the system invariants. For instance, an addition of a method depends on the addition of a class as you cannot add a method to a nonexistent class. As such a software system and its entire evolution is represented as a graph with the changes as nodes and the dependencies between the changes as edges.

Once the sequence of changes and their dependencies is recorded, we use Groove [24], a graph transformation tool, to search the change graph for pre-defined patterns corresponding to a refactoring. We chose Groove because it uses a simple XML format to store their graphs, as such it was easy to export the change graphs from ChEOPSJ into a Groove readable format. Besides that, Groove offers a fast and scalable state space exploration so it should be able to find our refactoring patterns on large graphs relatively quickly.

As an example, we briefly describe how we reconstruct a PullUpMethod refactoring from a graph of changes. The other refactorings are defined in a similar way and are published on **figshare** [29]. The Groove graph transformation is shown in Figure 3. The top of this pattern describes how the classes are related. The class from which the method is removed needs to be different from the class in which we added a method. The class node on the left should be a descendant to the class on the right. We express this relationship using a regular expression `(-superclass.subclass)+`, meaning that we traverse the edge in the opposite direction (with `-superclass`) from the superclass node to the implicit Inheritance node, and then traverse the edge in the normal direction (`subclass`) to the subclass node. Adding the `+` makes this the transitive closure, meaning that we can trace this edge to any descendant class in the inheritance tree of the superclass. The bottom half of the pattern describes the changes to the methods. In the subclass the method has (at least) two changes: an addition (which is dependent on the addition of the class) and a removal which is dependent on the addition of the method. In the superclass the method has (at least) one change: an addition. Moreover the method in the subclass and the superclass should have an identical name. The left side of the pattern (the subclass and its method along with the additions of both and the removal of the method) have a universal quantifier (∀) meaning that this pattern applies to all subgraphs of this kind. In other words, for each instance of a removal of the method in a subclass, this removal is part of the PullUpMethod refactoring reconstructed. This graph transformation rule adds a new node – PulledUpMethod – linked to (i) the changes that remove instances of the method in subclasses (ii) the change that adds the method to the superclass. However this should only be done if these changes are not already linked to a previously reconstructed refactoring node.

We argued the feasibility of a change-based approach for refactoring reconstruction in a previous paper where we had implemented a way for detecting MoveMethod and RenameMethod [35]. For this paper, we extended this proof by construction to incorporate 11 of the refactoring rules that can be expressed on the model in Figure 1 and Figure 2.

- PullUpMethod
- PullUpField
- PushDownMethod
- PushDownField
- MoveClass
- MoveMethod
- MoveField
- RenamePackage
- RenameClass
- RenameMethod
- RenameField

With this list, we have a sufficient basis to compare against the state-of-the-art snapshot-based tool RefFinder [14].

3. REFACTORING MASKING

In this section we address **RQ 1:** *Under which conditions does a snapshot-based approach fail to reconstruct refactorings?* We illustrate the refactoring masking phenomenon, by using the state of the art tool RefFinder [14] on a small yet realistic system: the LAN Simulation [6][2]. This is a script of refactorings that are performed on a small system. It is mostly used as a teaching lab to teach how and why to refactor.

3.1 Experimental Setup

We followed the script of the LAN Simulation [6] and injected some non-refactoring changes along the way. After every atomic change we committed revisions to a local subversion repository. For these commits, we handled the same level of granularity (method level changes) as the model in our change recording tool shown in Figure 2. For instance when performing a MoveMethod refactoring, we executed a simple copy, paste and delete and then updated the signature and the invocations. This resulted in at least 5 commits to the repository: after the copy; after the paste; after the delete; after the signature update; and after the invocation update. As such we created a fine grained change model stored in a subversion repository, with each revision containing one change. We then had RefFinder compare each revision with every other revision in order to find both the smallest and the largest distance between revisions needed to reconstruct a refactoring.

3.2 Results

We performed a series of 22 refactorings in 150 commits: 2 instances of IntroduceExplainingVariable; 7 instances of ExtractMethod; 10 instances of MoveMethod; 2 instances of ExtractSubclass and a single instance of ReplaceConditionalWithPolymorphism). The repository is published on **figshare** [30]. We compared all possible pairs of these 150 commits with RefFinder. That is, we compared revision 1 to revisions 2 to N, then we compared revision 2 to revisions 3 to N, and so on. We looked at the refactorings that RefFinder reconstructed all together, and summed up all the unique distinct refactorings it could reconstruct in all pairs of revisions. We found that RefFinder reconstructed 100 refactorings, of which 40 were false positives, 19 were true positives and 41 were neither, but could be considered as side effects (or subrefactorings) of the performed refactorings. For instance, a MoveMethod usually also involved the removal of a parameter, as the class to which the method was moved used to be a parameter. Another example was the ExtractSubClass refactorings, that also consisted of 3 MoveMethod refactorings. Additionally there were three false negatives: two instances of ExtractMethod and one instance of MoveMethod refactorings that we performed, but that RefFinder did not manage to reconstruct.

The important thing to note is that most of the refactorings we performed were reconstructable by RefFinder at one point or another. In Figure 4 we show the minimum and maximum windows under which these occur. The maximum window is shown below the minimum window; the latter shows the revisions where the refactoring was actually performed. The maximum window denotes up to which point, either before or after the minimum window boundaries, RefFinder is capable of identifying the performed refactoring. To the right of the figure, each refactoring is numbered for easy referencing in the following paragraphs.

We see that refactorings 7, 8, 13, 16, 17, 18, 19 and 20 have a window that reaches to the HEAD revision, which implies that these refactorings were not masked by any other changes. The other eleven refactorings are at one point masked by other changes, hence no longer reconstructable.

The first refactoring masking instances that we want to highlight are the refactorings 3, 6, 10, 12 and 14. These are one ExtractMethod and four MoveMethod refactorings that are masked by changes other than refactoring operations. The ExtractMethod was no longer reconstructable, since we completely changed the code inside the extracted method. It was changed in such a way that it semantically did more or less the same thing, but syntactically it was completely different. A similar observation can be made with the four MoveMethod refactorings that are masked by non refactoring changes.

The other six refactoring instances are masked by other refactoring operations (indicated with the dashed lines). The first are the two IntroduceExplainingVariable refactorings (refactorings 1 and 2), these were hidden by MoveMethod 3 (refactoring 8), as this move operation moved the method in which the variables were introduced. The figure then also shows that four ExtractMethod refactorings were hidden by a MoveMethod refactoring. What happened is some code was extracted to a method and this newly created method was then moved to a different class, at which point it was no longer possible to reconstruct the `ExtractMethod` refactorings. A special case is Extract Method 2, which is shown twice in the figure (as refactorings 4 and 5). This is because this refactoring extracted a duplicate series of statements from two methods into a new method. RefFinder identified this as two distinct ExtractMethod refactorings.

We conclude from this experiment that the minimum window in which a refactoring can be identified needs to be comprised of, at least, the changes of that refactoring. The maximum window in which the refactoring can be reconstructed is uncertain, as a refactoring can be hidden by other operations. In our case study, we have observed that a refactoring can always be reconstructed as long as no other changes act on the same source code entities as that refactoring. One could argue that it is possible to write new rules for RefFinder that identifies a combination of refactorings, but this is an infeasible approach since we can not be expected to devise rules for every possible combination of refactoring operations.

*As an answer to **RQ 1**, we conclude that a snapshot-based approach fails to reconstruct refactorings when other edit operations act on the same source code entities. We then say that the refactoring is* "masked" *by those other changes.*

4. COMPARING REFFINDER TO CHEOPSJ

In this section we address **RQ 2:** *Do fine-grained changes allow us to reconstruct refactorings where snapshot-based approaches fail?* We search for instances of refactoring masking in two real-world cases and see whether a change-based approach is capable of reconstructing refactorings where snapshot-based approaches do not.

[2] `http://lore.ua.ac.be/Research/Artefacts/refacLab/`

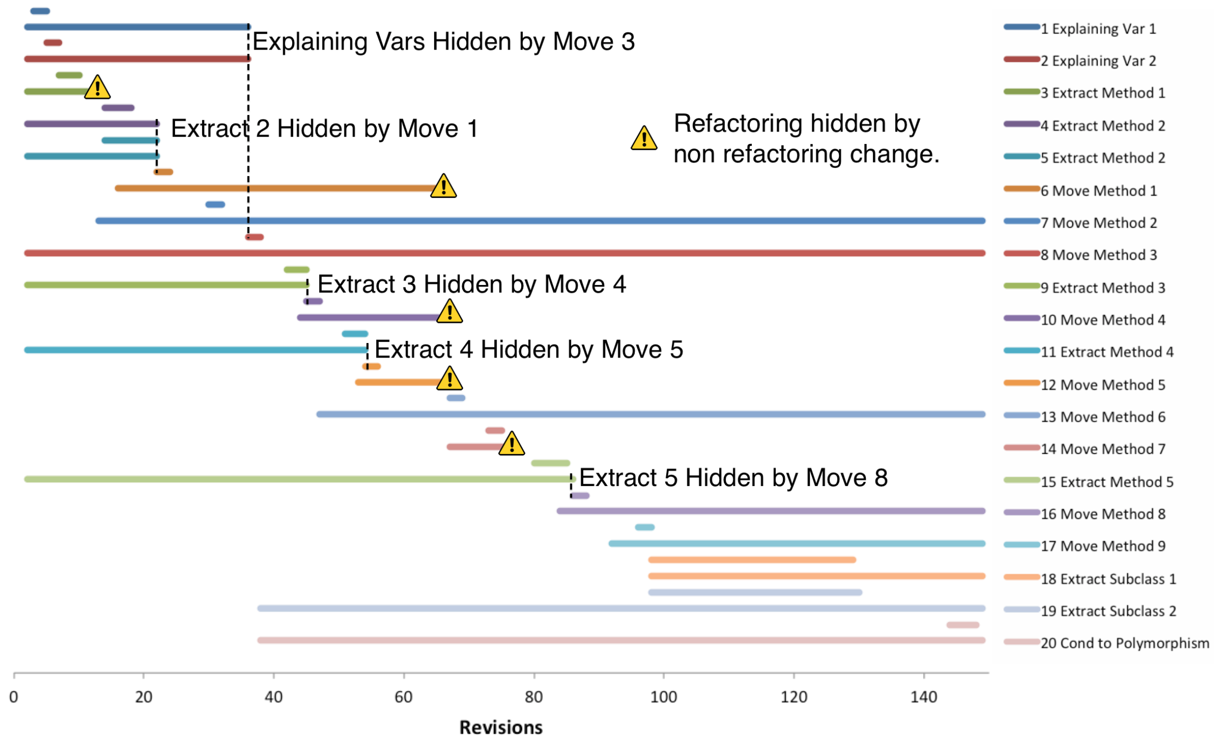

Figure 4: Minimum and maximum windows in which the performed refactorings are reconstructable by RefFinder.

4.1 Experimental Setup

We used two larger open source cases—PMD and Cruisecontrol. PMD[3] is a source code analyser to find a variety of common mistakes like unused variables, empty catch blocks, unnecessary object creations, and so on. Cruisecontrol[4] is a framework that allows for creating a custom continuous integration process.

These projects were selected as they are written in Java and their development history is freely available on a subversion repository. More importantly, the developers of these two software projects used the commit messages to consciously document some of the applied refactorings. We could thus mine these commit messages using a simple grep command to identify those revisions with documented refactorings. We searched for commit messages that contain the terms "refactor(-ed,-ing)", "move(-d)" or "rename(-d)".

We sampled these two projects for cases of refactoring masking and selected 10 revisions containing masked refactorings. Tables 1 and 2 show the selected revisions in which we have a total of 26 masked refactorings. These revisions were selected with the following criteria: each revision should be one with refactoring documented in the commit message; RefFinder should be unable to reconstruct any refactorings; and a manual analysis of the revision should show that there were actual refactorings performed and that RefFinder's refactoring reconstruction failed due to refactoring masking. Note that this is a very conservative way of looking for cases of refactoring masking, since we cannot claim that there is no refactoring masking in either the revisions that had no documented refactorings or the revisions where RefFinder did find refactorings.

[3]`http://pmd.sourceforge.net`
[4]`http://cruisecontrol.sourceforge.net`

In these cases of refactoring masking, we set out to re-perform those refactorings while recording our changes with ChEOPSJ in order to obtain a fine grained change model. Suppose revision R_x is documented as a refactored version of revision R_{x-1} and RefFinder cannot identify the performed refactorings because they were masked by other changes. In this case we checked out revision R_{x-1} and distilled an addition change for every source code entity in this revision to have a starting set of changes. We then started recording the changes and performed the refactorings that we identified during our manual analysis in order to obtain the fine-grained change history. At the end of re-performing the refactorings, we verified (using Eclipse's built in compare support) that our changed version of the system matched revision R_x. We could then export our graph of recorded changes to a Groove readable format and had Groove perform our graph transformation rules to reconstruct the refactoring instances.

4.2 Refactoring Masking Instances

In the PMD project we analysed 6 revisions, where RefFinder was unable to reconstruct the refactorings performed due to refactoring masking. The details of these six revisions are in Table 1. In all of the cases, the masking involved an ExtractMethod refactoring followed directly by a MoveMethod or PullUpMethod refactoring. This means that the developers extracted a piece of code and immediately moved it to a class where they felt it belonged. RefFinder is unable to reconstruct either of these refactorings. For the ExtractMethod it looks for a newly created method in the same class, but the method is already moved to a different class. For the MoveMethod it looks for the class from which the method originates, but there was no such method to begin with.

Table 1: Refactoring masking in PMD.

Rev.	Commit Message	Refactorings Performed
578	**refactor**ing	ExtractMethod MoveMethod
659	more CPD-aided **refactor**ing	ExtractMethod PullUpMethod 2xInlineTemp
1082	more **refactor**ing and tests	ExtractMethod MoveMethod InlineTemp
1085	more **refactor**ing	ExtractMethod MoveMethod
1088	minor **refactor**ing	ExtractMethod MoveMethod RenameLocalVar
2207	**Refactor**ed some more code into `CommandLineOptions`, wrote some more tests	ExtractMethod MoveMethod
	Number of refactorings masked	16

A typical example is shown in listings 1 and 2. An ExtractMethod extracted some code from the `visit` method in the class `EmptyFinallyBlockRule` into a new method called `getFinallyBlock`. This method was then moved to the class `ASTTryStatement`.

```
package net.sourceforge.pmd.rules;
public class EmptyFinallyBlockRule extends AbstrRule {
  public Object visit(...){
    //some Code that is extracted and moved
  }
}

package net.sourceforge.pmd.ast;
public class ASTTryStatement extends SimpleNode {
}
```

Listing (1) PMD revision 1084.

```
package net.sourceforge.pmd.rules;
public class EmptyFinallyBlockRule extends AbstrRule {
  public Object visit(...){
    ASTBlock finallyBlock = node.getFinallyBlock();
  }
}

package net.sourceforge.pmd.ast;
public class ASTTryStatement extends SimpleNode {
  public ASTBlock getFinallyBlock() {
    //some Code that is extracted and moved
  }
}
```

Listing (2) PMD revision 1085.

In the case of Cruisecontrol we analysed 4 revisions containing instances of refactoring masking. We detailed these revisions in Table 2. One of these (rev 842) is similar to the refactoring masking instances in PMD, in that a new method was extracted and immediately pushed to the responsible subclass. The one difference is that the extracted method had the same identifier as the method from which it was extracted and the class to which it was moved was a subclass. As such they created a polymorphic version of the same original method. To maintain behaviour and thus make it a pure refactoring, they added a call to the super implementation. Another similar operation was in revision 2257, where they moved a few statements from one method to another method in a different class. One way of doing this is simply cutting and pasting the code from the one method to the other, which is probably what the developers did. However this same result could also be achieved by performing an ExtractMethod and a MoveMethod to get the piece of code to the right class. Then the invocation to the new method needs to be removed from the original method and an invocation needs to be added in the place where we want the code. To finish off an InlineMethod would put the code where we want it.

Table 2: Refactoring masking in Cruisecontrol.

Rev.	Commit Message	Refactorings Performed
842	Making `buildResultsUrl` optional in `HTMlEmailPublisher`. **Move**d validation of the parameter from `EmailPublisher` to `LinkEmailPublisher`.	Combination of ExtractMethod PushDownMethod RenameMethod
879	**rename**d `_now` and `getNow()` to `timeOfCheck` and `getTimeOfCheck()`	RenameMethod RenameField
2257	**move** knowledge of default log location into `Log` class	Combination of ExtractMethod MoveMethod InlineMethod
2629	... Also **rename**d `IO.deleteFile` to `IO.delete` because the File part is obvious from the signature and therefor superfluous....	RenameMethod RenameParameter
	Number of refactorings masked	10

Cruisecontrol also provided us with a few examples of masked RenameMethod refactorings. In one case (revision 879), this refactoring is hidden by a RenameField refactoring. In RefFinder the rule to reconstruct a RenameMethod checks whether the method body has a certain similarity. In this case the method body consisted of a single statement: a return statement returning the value of the field. Since the field itself was also renamed to a name that is very different, the method body was no longer similar enough to count as a RenameMethod.

4.3 Change-Based Reconstruction

Our approach for refactoring reconstruction based on recorded changes was capable of reconstructing all refactorings for which we currently have rules. More precisely, our approach allowed us to identify 12 out of 26 refactorings that, for RefFinder, were masked.

Specifically we could reconstruct all of the MoveMethod refactorings and the one PullUpMethod refactoring that we performed in PMD as well as the two MoveMethod, the two RenameMethod and the one RenameField refactorings we performed in Cruisecontrol. As an example we present the results of the refactoring pattern reconstructed from the changes we performed to go from revision 658 to revision 659 in PMD (Figure 6) and the patterns reconstructed from the changes between revision 878 and 879 in Cruisecontrol (Figure 7). All other resulting graphs can be

found on **figshare** [31]. Figure 6 shows that there was a PULLUPMETHOD refactoring that removed two instances of a method named "`getEndName`" in two subclasses of the class "`UnusedCodeRule`" and an addition of a method by the same name in the superclass. Figure 7 shows that in Cruisecontrol we were able to reconstruct two refactorings from a set of changes: one is the RENAMEFIELD refactoring that removes the attribute `_now` and adds the attribute `timeOfCheck`; and the RENAMEMETHOD refactoring that changes the name of this attribute's getter from `getNow` to `getTimeOfCheck`.

*We conclude that the presence of fine-grained changes allows us to reconstruct refactorings where snapshot-based approaches fail. Indeed, we have found several occurrences of masked refactorings in two real-world open sourced cases, all of which RefFinder was unable to reconstruct. In contrast, our change-based approach was able to reconstruct 12 out of 26 masked refactorings; the other refactorings could be reconstructed by extending the source code model (see Figure 2) and defining new rules for these particular refactorings. As such we effectively answered **RQ 2**.*

4.4 Is this relevant?

Knowing that change-based approaches are capable of reconstructing refactorings where snapshot-based approaches fail, the natural follow up question is to what extent is this improvement relevant. That is, how often does refactoring masking occur in real software projects? This question is impossible to answer precisely, given that there is no project where all refactoring operations are recorded [17]. Moreoever, Negara *et al.* already reported that on average 30% of refactoring operations do not reach the version control system [18]. Nevertheless, we can make a rough estimate based on the data gathered by the Eclipse Usage Data Collector (UDC)[5]. This data is made publicly available and Emerson Murphy Hill et. al. have put the whole data set on Google BigQuery, which enables us to query and process this data using Google's storage and compute infrastructure [28].

We know that snapshot-based approaches typically fail when several edit operations act on the same source code entities. In particular, sequences of EXTRACTMETHOD, MOVEMETHOD (and in the case of Cruisecontrol also a RENAMEMETHOD) operating on the same segments of code are likely to cause misses. From the UDC dataset it is clear that the RENAMEMETHOD operation is by far the most used automated refactoring in Eclipse; MOVE–ELEMENT– and EXTRACTMETHOD are also among the top most used automated refactoring operations. This gives a first indication that refactoring masking is relevant.

We are most interested in the combination of EXTRACTMETHOD and MOVEMETHOD, so we looked at the edit operations that occurred within 5 minutes after an EXTRACTMETHOD operation (see Figure 8). Here we assume programming locality, that is two edit operations that occur closely together are likely to act on the same source code entities [23]. MOVE–ELEMENT– appears at the nineteenth position, but the top three operations are DELETE, PASTE and COPY; which serve as a manual substitute for a move. Launching a query which counts all EXTRACTMETHOD operations that are followed within 5 minutes by either a MOVEMETHOD or by a COPY, a PASTE and a DELETE, we found 10,869 instances out of a total of 43,602, thus almost 25%. Note that this is a conservative estimate, as these are the situations where we know a snapshot-based approach will fail. In reality, there are likely to be more, since we only took into account those change sequences which start with an extract method. Therefore, we argue that a snapshot-based approach is likely to miss a significant amount of the refactoring sequences, hence that the potential improvements induced by a change-based approach are indeed relevant.

[5]http://www.eclipse.org/epp/usagedata

5. THREATS TO VALIDITY

We now identify factors that may jeopardise the validity of our results and the actions we took to alleviate the risk. Consistent with the guidelines for case studies research [26, 40] we organise the identified threats into four categories.

Construct validity – do we measure what was intended. We relied on the versioning system's log messages to identify revisions corresponding to refactorings. Since no strict conventions are in place for what should be specified in such messages, there may be significant differences in the content and quality of log messages across tasks and developers. Consequently, we might miss certain revisions which do correspond to refactorings. However it was never our intent to find all instances of refactorings that occurred in the system's evolution. In that sense, using this simple way of locating instances of refactorings is sufficient for our purposes.

An additional threat could be that the expertise and experience of developers plays a key role in this application. A developer that knows the purpose of the different refactorings, might be less inclined to perform floss refactoring and actually commit the refactoring as a whole to the source code management system.

Moreover, the fine-grained recorded changes did not exist in the original sample from the repositories so we had to manually re-perform them. These changes might not be the ones applied by the developers. The transformations we performed, in order to obtain the fine-grained change history, are just one possible change-sequence from many potential scenarios. We used our expertise to choose the transformations that we would have applied. We verified (using Eclipse's built in compare support) that our changed version of the system matched the actual revision in the repository.

Internal validity – are there unknown factors which might affect the outcome of the experiment. The substitution of "real" recorded changes by a manual synthetic reproduction of them might be a confounding factor for our results. The improvement in the number of masked refactorings detected by our approach over RefFinder might not be due to the availability of fine-grained changes but to the particular change sequences we manually applied. To reduce this risk, the first two authors of this paper proposed and discussed these change sequences in order to come up with the transformations that, in our opinion, are closest to the real ones.

As already mentioned, there can also be a problem of selection bias caused by how we decided on the refactored revisions we used as experiment subjects. The need to verify

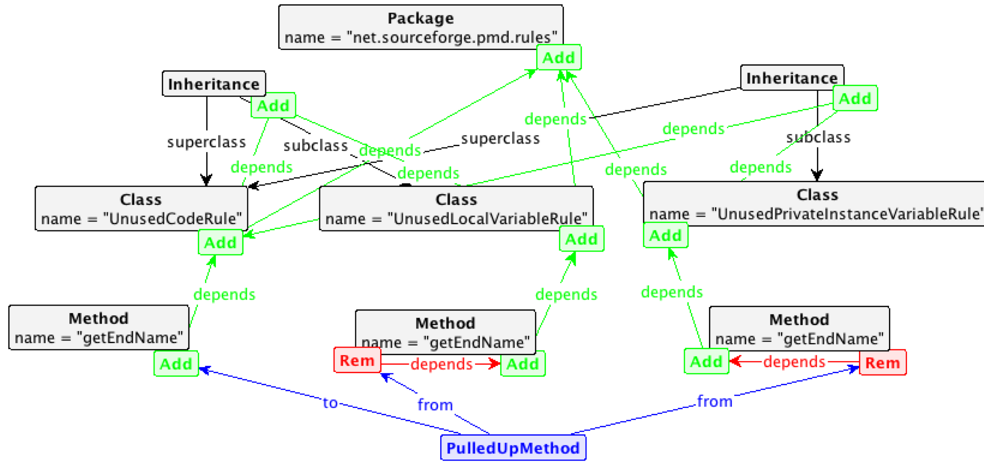

Figure 6: The PullUpMethod refactoring identified by ChEOPSJ in PMD. (note: The subject relationships between changes and source code entities are hidden for the sake of readability.)

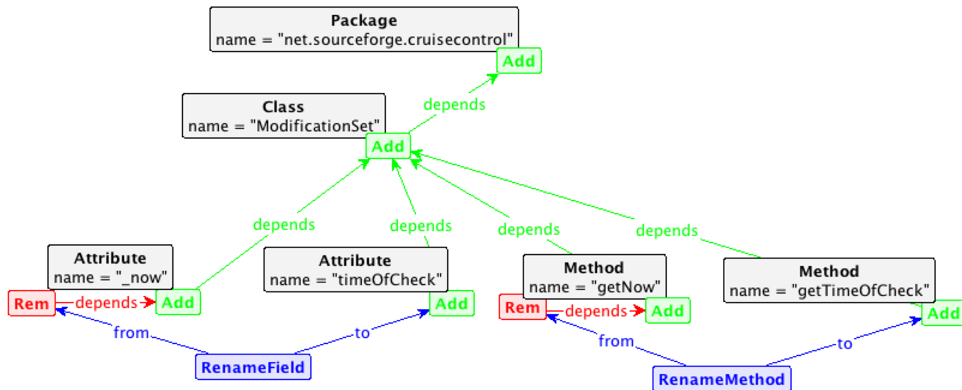

Figure 7: The RenameMethod and RenameField refactorings identified by ChEOPSJ in CruiseControl. (note: The subject relationships between changes and source code entities are hidden for the sake of readability.)

RefFinder results and to manually apply the fine-grained changes, led us to sample only those revisions whose commit messages mentioned refactoring operations. In order to alleviate the potential bias, we run the experiments in a toy example and in two different open source systems. Similar results were obtained from them.

External validity – to what extent is it possible to generalise the findings. In this study we investigated two cases: Cruisecontrol and PMD. We chose them to be sufficiently different, yet, with only two data points, we cannot claim that our results generalise to other systems. The results are also dependent on the number of refactoring detection rules we have implemented. The instances of refactoring masking we have analysed might very well appear in other systems. We cannot however make any claims about other types of (as yet unidentified) refactoring masking.

According to [19], changes in snapshot-based versions are often obscured by other changes. We can expect the refactoring masking problem to be more prominent than what we have inspected. We should also expect different behaviours when developers commit to SVN or Git. It has been observed that programmers commit more often to Git repositories, resulting also in smaller commits [2]. The need for a change-recording mechanism might not be so important in the context of Git repositories, although it also depends on whether the developers use commit squashing (grouping several related changes in one single commit).

Reliability – is the result dependent on the tools. We used RefFinder to construct the baseline for our experiments. The tool might have produced false positives and false negatives, wrongly identifying some refactoring while missing others. On the one hand, this is the best existing tool for refactoring detection. Therefore, despite of the possible errors, it still serves well as a baseline to compare with. On the other hand, we have manually verified the refactorings detected by RefFinder thus, reducing the risk of false positives.

In order to implement our approach, we relied on tools of our own making as well as some external tools. Our ChEOPSJ tool is implemented as an Eclipse plugin and relies on Eclipse's internal java model which can be considered to be a reliable tool. In order to reduce the bias caused by possible bugs and errors in the tool, we tested it extensively. Members of the research groups and some developers at a partner company installed ChEOPSJ and acted as beta-testers for three months. This resulted in many bug fixes

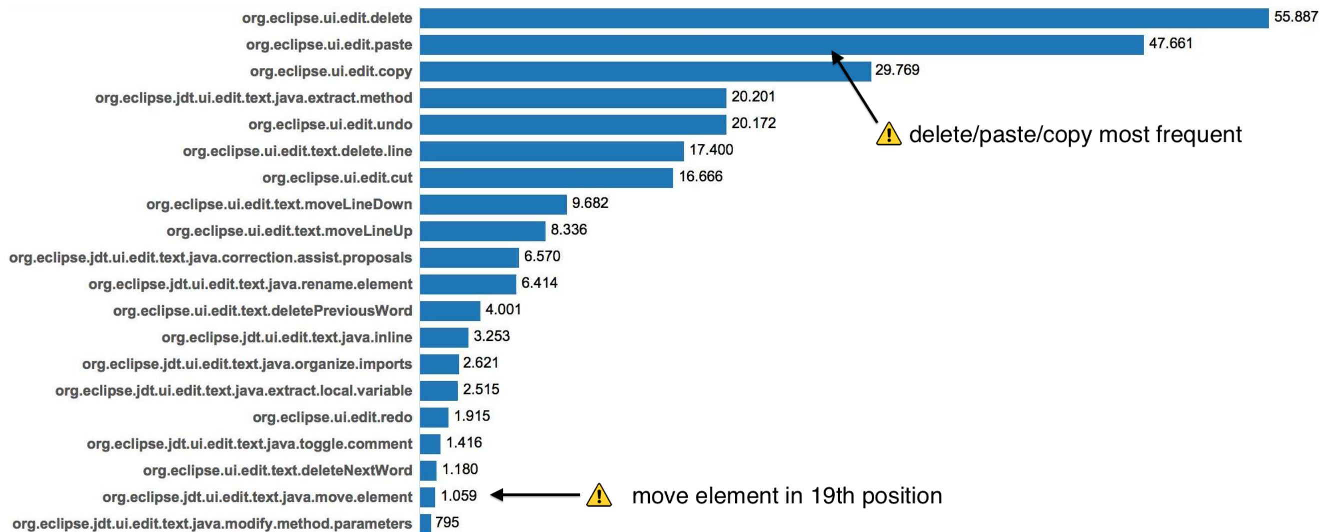

Figure 8: Top 20 most used edit commands within 5 minutes after an ExtractMethod.

and led to a stable and robust version of the tool. For the graph pattern matching we used the tool Groove, which is still actively being developed and improved, but since it has already evolved to a stable and robust tool it too can be considered reliable.

6. CONCLUSIONS

In this paper we have shown how a software evolution history comprised of fine-grained recorded changes can be exploited to reconstruct refactorings more accurately than the state-of-the-art snapshot-based technique RefFinder. To provide a more detailed summary of our findings, we review the research questions we have addressed:

RQ 1 *Under which conditions does a snapshot-based approach fail to reconstruct refactorings?* Snapshot-based approaches fail when other edit operations act on the same source code entities. In particular, combinations of ExtractMethod and MoveMethod confuse a snapshot-based approach, since the effect of the former is concealed by the latter. We then say that the refactoring is *"masked"* by those other changes. Since such simultaneous edit operations might happen at any time, it is impossible to determine the optimal window of changes where snapshot-based reconstruction will still function properly. Hence, the only alternative to faithfully reconstruct refactorings is to have access to the fine-grained changes applied to the code.

RQ 2 *Do fine-grained changes allow us to reconstruct refactorings where snapshot-based approaches fail?* We sampled the version history of two open source projects where the developers made an effort to explicitly document some of the refactorings applied. In particular, we made an opportunistic sample, selecting versions where simultaneous edits on the same entity were performed. Under these conditions, snapshot-based approaches indeed fail to reconstruct the refactorings, while the change-based approach does succeed. Next, we argued that these conditions occur frequently: we estimate that for 25% of all the ExtractMethod refactorings are followed by either a MoveMethod or by a Copy, a Paste and a Delete

Contributions. Over the course of this research, we made the following contributions:

- We implemented a tool prototype named ChEOPSJ serving as an experimental platform for conducting feasibility studies with first-class representation of changes in Java.
- We demonstrated how this platform can be used to reconstruct refactoring operations from a stream of changes.
- We applied the prototype to two cases — PMD and CruiseControl — to compare a change-based approach against a snapshot-based approach.
- We demonstrated that the change-based approach is more accurate than a snapshot-based approach.
- We argued that this improved accuracy is relevant by estimating the number of edit operations acting on the same source code entities within 5 minutes after an ExtractMethod refactoring.

Future work. Our plan for the immediate future is to gather real recorded data and replicate our experiments, thus moving from In-Vitro to In-Vivo research [4]. We are currently deploying the change-recording plugin at some partner companies which should result in detailed streams of changes where we can interview the developers about the details of the refactorings. Next we plan to implement additional detection rules for other refactorings besides the 11 listed under Section 2.2, to investigate other refactoring masking conditions. Particularly, interesting in that respect would be a more detailed change model fully representing AST entities below the method signature level.

Our findings together with similar results obtained by others (i.e., Spyware [25], Syde [12], Cheops [9], OperationRecorder [20]), indicate that it is not only feasible but also worthwhile to maintain fine-grained evolution histories of software projects. Given the popularity of Git, which encourages a fine-grained commit behaviour, having an explicit representation of the changes is the natural successor to the current generation of distributed version control systems.

7. ACKNOWLEDGMENTS

This work has been sponsored by (i) the Interuniversity Attraction Poles Programme - Belgian State - Belgian Science Policy, project MoVES; (ii) the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen) under project number 120028 entitled "Change-centric Quality Assurance (CHAQ)". We hereby express our gratitude to Arend Rensink for his quick response on the Groove discussion forum[6], as such helping us out with the Groove Syntax.

8. REFERENCES

[1] A. Abran, P. Bourque, R. Dupuis, J. W. Moore, and L. L. Tripp. *Guide to the Software Engineering Body of Knowledge - SWEBOK*. IEEE Press, 2004.

[2] C. Brindescu, M. Codoban, S. Shmarkatiuk, and D. Dig. How do centralized and distributed version control systems impact software changes? In *Proceedings ICSE, 2014*, pages 322–333.

[3] S. Demeyer, S. Ducasse, and O. Nierstrasz. Finding refactorings via change metrics. In *Proceedings OOPSLA, 2000*, pages 166–177.

[4] S. Demeyer, A. Lamkanfi, and Q. D. Soetens. "in vivo" research in software evolution. *ERCIM News*, 2012(88), 2012.

[5] S. Demeyer, S. Tichelaar, and P. Steyaert. FAMIX 2.0 - the FAMOOS information exchange model. Technical report, University of Berne, 1999.

[6] S. Demeyer, F. Van Rysselberghe, T. Girba, J. Ratzinger, R. Marinescu, T. Mens, B. Du Bois, D. Janssens, S. Ducasse, M. Lanza, M. Rieger, H. Gall, and M. El-Ramly. The LAN-simulation: a refactoring teaching example. In *Proceedings IWPSE, 2005*, pages 123–131.

[7] D. Dig, C. Comertoglu, D. Marinov, and R. Johnson. Automated detection of refactorings for libraries and frameworks. In *Proceedings WOOR, 2005*

[8] D. Dig, C. Comertoglu, D. Marinov, and R. Johnson. Automated detection of refactorings in evolving components. In *Proceedings ECOOP, 2006*, pages 404–428

[9] P. Ebraert, J. Vallejos, P. Costanza, E. V. Paesschen, and T. D'Hondt. Change-oriented software engineering. In *Proceedings ICDL, 2007*, pages 3–24.

[10] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 2001.

[11] C. Görg and P. Weißgerber. Error detection by refactoring reconstruction. In *Proceedings MSR, 2005*

[12] L. Hattori and M. Lanza. Syde: A tool for collaborative software development. In *Proceedings ICSE, 2010*, pages 235–238.

[13] J. Henkel and A. Diwan. CatchUp!: capturing and replaying refactorings to support API evolution. In *Proceedings ICSE, 2005*, pages 274–283.

[14] M. Kim, M. Gee, A. Loh, and N. Rachatasumrit. Ref-finder: a refactoring reconstruction tool based on logic query templates. In *Proceedings FSE, 2010*, pages 371–372.

[15] H. Liu, Y. Gao, and Z. Niu. An initial study on refactoring tactics. In *Proceedings COMPSAC, 2012*, pages 213–218.

[16] A. Murgia, M. Marchesi, G. Concas, R. Tonelli, and S. Counsell. Parameter-based refactoring and the relationship with fan-in/fan-out coupling. In *Proceedings. ICST Workshops, 2011*, pages 430–436.

[17] E. Murphy-Hill, C. Parnin, and A. P. Black. How we refactor, and how we know it. *IEEE TSE*, 38(1):5–18, 2012.

[18] S. Negara, N. Chen, M. Vakilian, R. Johnson, and D. Dig. A comparative study of manual and automated refactorings. In G. Castagna, editor, *ECOOP, 2013*, volume 7920 of *LNCS*, pages 552–576.

[6] `http://sourceforge.net/p/groove/discussion/407076/thread/8eb1f2c4/`

[19] 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 *Proceedings ECOOP, 2012*, pages 79–103.

[20] T. Omori and K. Maruyama. A change-aware development environment by recording editing operations of source code. In *Proceedings MSR, 2008*, pages 31–34.

[21] K. Prete, N. Rachatasumrit, N. Sudan, and M. Kim. Template-based reconstruction of complex refactorings. In *Proceedings ICSM, 2010*, pages 1–10.

[22] N. Rachatasumrit and M. Kim. An empirical investigation into the impact of refactoring on regression testing. In *Proceedings ICSM, 2012*, pages 357–366.

[23] V. Rajlich. Modeling software evolution by evolving interoperation graphs. *Ann. Softw. Eng.*, 9(1-4):235–248, Jan. 2000.

[24] A. Rensink. The groove simulator: A tool for state space generation. In *AGTIVE, 2004*, volume 3062 of *LNCS*, pages 479–485.

[25] R. Robbes and M. Lanza. SpyWare: A change-aware development toolset. In *Proceedings ICSE, 2008*, pages 847–850.

[26] P. Runeson, M. Höst, and M. Alshayeb. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, 2009.

[27] C. Schofield, B. Tansey, Z. Xing, and E. Stroulia. Digging the development dust for refactorings. In *Proceedings ICPC, 2006*, pages 23–34.

[28] W. Snipes, E. Murphy-Hill, T. Fritz, M. Vakilian, K. Damevski, A. R. Nair, and D. Shepherd. *Analyzing Software Data*, chapter A Practical Guide to Analyzing IDE Usage Data. Morgan Kaufmann, 2015.

[29] Q. D. Soetens. Groove Refactoring Reconstruction Rules, 06 2014, Retrieved 09:48, Jul 06, 2015 (GMT), **figshare** http://dx.doi.org/10.6084/m9.figshare.1070082

[30] Q. D. Soetens. LAN Simulation Fine Grained Changes Repository, 11 2014, Retrieved 09:49, Jul 06, 2015 (GMT), **figshare** http://dx.doi.org/10.6084/m9.figshare.1237061

[31] Q. D. Soetens. Results for Reconstruction on Cruisecontrol and PMD, 06 2014, Retrieved 09:41, Jul 06, 2015 (GMT), **figshare** http://dx.doi.org/10.6084/m9.figshare.1080418

[32] Q. D. Soetens and S. Demeyer. Studying the effect of refactorings: a complexity metrics perspective. In *Proceedings QUATIC, 2010*, pages 313–318.

[33] Q. D. Soetens and S. Demeyer. ChEOPSJ: Change-based test optimization. In *Proceedings CSMR, 2012* pages 535–538.

[34] Q. D. Soetens, S. Demeyer, and A. Zaidman. Change-based test selection in the presence of developer tests. In *Proceedings CSMR, 2013* pages 101–110.

[35] Q. D. Soetens, J. Pérez, and S. Demeyer. An initial investigation into change-based reconstruction of floss-refactorings. In *Proceedings ICSM, 2013*, pages 384–387.

[36] K. Stroggylos and D. Spinellis. Refactoring–does it improve software quality? In *Proceedings WoSQ, 2007*, page 10.

[37] F. Van Rysselberghe and S. Demeyer. Reconstruction of successful software evolution using clone detection. In *Proceedings IWPSE, 2003*, pages 126–130.

[38] P. Weißgerber and S. Diehl. Are refactorings less error-prone than other changes? In *Proceedings MSR, 2006*, pages 112–118.

[39] Z. Xing and E. Stroulia. Refactoring detection based on UMLDiff change-facts queries. In *Proceedings WCRE, 2006*, pages 263–274.

[40] R. K. Yin and M. Alshayeb. *Case Study Research: Design and Methods, 3 edition*. Sage Publications, 2002.