in Proceedings of 21st IEEE/ACM International Conference on Automated Software Engineering (ASE 2006) , Tokyo, Japan, 2006 (received best paper award).

Identifying Refactorings from Source-Code Changes

Peter Weißgerber
Computer Science
University of Trier
54286 Trier, Germany
cs@p-weissgerber.de

Stephan Diehl
Computer Science
University of Trier
54286 Trier, Germany
diehl@acm.org

Abstract

Software has been and is still mostly refactored without tool support. Moreover, as we found in our case studies, programmers tend not to document these changes as refactorings, or even worse label changes as refactorings, although they are not. In this paper we present a technique to detect changes that are likely to be refactorings and rank them according to the likelihood. The evaluation shows that the method has both a high recall and a high precision — it finds most of the refactorings, and most of the found refactoring candidates are really refactorings.

1. Introduction

Having been absent from a software project for a while, programmers sometimes have a hard time to find their previous contributions to the source code. Wouldn't it be nice, if someone told them, that all what happened to their code were just a few refactorings.

Although various tools have been developed to automatically apply refactorings and even to record these, most systems have been and are still refactored without such tools and often without the developer even knowing that what she is doing is a refactoring.

Extracting refactorings from software archives is a prerequisite for many applications:

Detecting Possible Sources of Errors To preserve the program behavior refactorings often require changes at several locations in the source code. As shown in a previous paper [9] extracted refactoring candidates can be checked for completeness, i.e., whether all related locations have been changed. Incomplete refactorings can lead to compilable programs with wrong behavior.

Based on the information in bug databases one can also relate fixes to program parts that have been refactored before and thus assess the error-proneness of different kinds of refactorings. Tool support should focus on error-prone refactorings.

In a first case study [17] we used the techniques presented here together with information from bug repositories to investigate the error-proneness of transactions with a high refactoring ratio.

Capturing Intent of Changes Changes can be classified in various ways, e.g., feature extensions or bug fixes. An important class of changes are refactorings. A refactoring is a high-level description of a change that emphasizes its intent. The change history of a piece of code could thus be described as a sequence of higher-level changes.

Complex program changes like platform migrations, adding of concerns like logging, or networking could be described in this way and compared between different systems, versions or parts of a system.

The information can also be useful when several developers work independently on the same part of a system and have to integrate their changes, or if a programmer has been absent from a project for a while.

Capturing and Replaying of Changes If the changes of a particular part of code are purely refactorings, then these refactorings might be applicable to similar or related program code. An Eclipse plugin called CatchUp! [10] captures refactorings performed with the refactoring tool included in Eclipse. The recorded refactorings of an API can thus be used to help users of the API to change their code, e.g., if a parameter was added to an API method, then method invocations in program code using the API have to be adapted. The same holds for frameworks.

In a similar way changes to the core system could be propagated to the various product lines or from one product line to others.

Relating to other Changes If the structure of the program code is changed by a refactoring, related documents like architectural designs or API documentations have to be kept consistent with the refactored version of the code. Combining refactorings extraction with evolutionary coupling (co-change information) can identify documents that

21st IEEE International Conference on Automated Software Engineering (ASE'06)
0-7695-2579-2/06 $20.00

are typically changed when a refactoring is applied.

In the ROSE [19] system we used information about evolutionary coupling to recommend program changes. Based on extracted refactorings the system could be improved: for move and rename refactorings the evolutionary coupling information of the previous instance and the new instance can be combined.

Relation to Software Metrics Software quality metrics can be computed for subsequent versions of a software system where the newer version mostly results by refactorings from its predecessor. This information could be used to assess what kinds of refactorings increase what kinds of quality metrics.

In this paper we present a novel technique to detect and rank refactoring candidates. In contrast to the state of the art (see Section 4) and our first attempts [8], this new technique has been thoroughly evaluated, and has demonstrated to have both a high recall and a high precision.

The remainder of this paper is organized as follows. Section 2 introduces our technique to detect refactoring candidates in a software archive and to rank them. In Section 3 we present an evaluation of the approach based on the software archives of two open source projects; we estimate the total number of refactorings and show the distribution of refactorings over the project's lifetime. Finally, Section 4 discusses related work and Section 5 summarizes and concludes the paper.

2. Identifying Refactorings

To detect refactorings in software archives, we first preprocess the repository and store the most important data in a relational database to get fast access to it. Then, we reconstruct transactions—that means which versions have been checked-in to the repository by the same commit operation. These transactions have to be analyzed for refactorings. For this, we first look for added, changed or removed entities (that are classes, fields, methods) to get refactoring candidates. Using clone detection we rank these candidates to indicate which are more likely to not have changed the behavior and thus are more likely to be real refactorings.

Currently, our technique is able to find refactorings of the following two general kinds (for many other refactoring kinds our signature-based approach should work as well):

Structural Refactorings include refactorings that change the class structure of the software. Our current prototype detects structural refactorings of the kinds *Move Class*, *Move Interface*, *Move Field*, *Move Method*, and *Rename Class*.

Local Refactorings are refactorings that are performed *within* a class. They include *Rename Method*, *Hide Method*, *Unhide Method*, *Add Parameter*, and *Remove Parameter*.

Most of these refactoring kinds are self-explanatory. *Hide method* means that the visibility of a method within a class has been changed to a more restrictive one, e.g. from `public` to `protected`. In contrast, making the visibility less restrictive is covered by the *unhide method* refactoring.

2.1. Pre-processing

Our approach to pre-process version data, i.e., data extraction and data cleansing, is described in detail in our earlier work [18] exemplarily for archives managed with CVS [3]. As a result of this step we gain the following information:

Versions. A *version* describes one revision of a file in the CVS repository (e.g., file `org/epos/epos.java` in revision 1.4). For each version in the repository we also store information about the committer, the log message, the branches the version belongs to, the timestamp of the check-in, the state (e.g., "dead" for actually deleted revisions), and the predecessor revision if one exists. The text, i.e. for program files the source code, of a version can be retrieved on demand.

Transactions. A *transaction* is the set of versions that have been committed to the repository at the same time by the same developer. Unfortunately, CVS splits commits that contain more than one file into single check-ins for each file and does not store which of these check-ins have been issued together. Thus, we use a sliding time window heuristic to recover transactions quite precisely. For every transaction we additionally store the timestamps of the *transaction start* and *end*.

CVS offers the feature to develop software in several parallel branches that live simultaneously to the main line of development which is often called *trunk*. In our work, we consider every transaction, no matter to which branch it belongs to, but we omit so-called merge transactions that incorporate all changes done in a branch into the parent development line. For our application it is reasonable to omit these merge transactions because they repeat refactorings that are actually done earlier in the respective branch. There exist few techniques to identify merge transactions [6, 20], but they are cumbersome to implement and their usefulness heavily depends on the merging policy of the project. Thus, we use the simple heuristic to classify all transactions that contain more than 50 versions as a merge.

2.2. Syntactical Analysis

Next we identify those software artifacts, e.g., packages, classes, or methods, that have actually been changed in a transaction. The actual change of such an artifact may be a refactoring.

We use a regular expression based on a light-weight parser that computes for each version $v \in V$ of a JAVA file the following sets:

The set C_v of classes: Its elements are pairs (p, n) where p is the name of the package the class belongs to, and n is the class name.

The set I_v of interfaces: Its elements are pairs (p, n) where p is the name of the package the interface belongs to, and n is the interface name.

The set M_v of methods: Its elements are tuples (c, m, p, r, w) where c is the fully-qualified name of the method's class, m is the method name, p is the parameter list, r is the return type, and w is the visibility of the method. The parameter list $p = [t_1, \ldots, t_k]$ enumerates the types of the k parameters of the method in the order as the parameters are defined. The visibility of a method can either be `public`, `protected` (visible to the package it belongs to, and to its subclasses), *default* (visible only to the package it belongs to), or `private`.

The set F of fields: Its elements are triples (c, f, t) where c is the fully-qualified name of the field's class, f is the name of the field, and t is the type of the field.

We call the elements in the above sets *entities*. Entities represent actual versions of software artifacts. The set $E_v = C_v \cup I_v \cup M_v \cup F_v$ is the set of all entities in version v and the set of all entities is $E = \bigcup_{v \in V} E_v$.

2.3. Signature-based Analysis

By definition, a refactoring may change the structure of a software system, but not its external behavior. In the next step of our process, we search for changes in the program code that correspond to refactorings. For example, for a refactoring of kind *add parameter* the code has to be changed as follows: To the signature of a method one or more parameters are added, but its name and its return type remain the same. If a method has been changed this way, it is a *refactoring candidate* because the above criteria does not suffice to determine whether its external behavior did or did not change.

Next, we formally describe the criteria of the signature-based analysis to find *refactoring candidates* for local as well as for structural refactorings.

Local refactorings

Local refactorings are refactorings that occur within one class, and thus, within the same file. To detect local refactorings that have been done in a transaction t, such as *rename method*, or *add/remove parameter*, we identify for each *newer version* $v' \in t$ its predecessor version v. This is simply done by looking which version has the same fully-qualified filename as v' and a version number that indicates that it is the predecessor version (e.g., for the file `A.java:1.4` the file `A.java:1.3` is the predecessor version).

Table 1. Conditions for Local Refactorings

Let $(c', m', p', r', w') \in M'$ be a method in the newer version. It consists of a class, a method name, a list of parameter types, a return type, and a visibility level. The method results from its previous version by one of the following refactorings, if the corresponding condition in the right column holds.

Add Parameter	$\nexists(c', m', p', r', *) \in M$, but $\exists(c', m', p, r', *) \in M$ and $p \sqsubset p'$
Remove Parameter	$\nexists(c', m', p', r', *) \in M$, but $\exists(c', m', p, r', *) \in M$ and $p \sqsupset p'$
Hide Method	$\exists\,(c', m', p', r', w) \in M$ and $w' \prec w$
Unhide Method	$\exists\,(c', m', p', r', w) \in M$ and $w \prec w'$
Rename Method	$\nexists(c', m', p', r', *) \in M$, but $\exists(c', m, p', r', *) \in M$ and $m \neq m'$

We define the order $\sqsubset$ on lists of types and the order $\prec$ on visibility levels as follows:
- $[t_1, ..., t_p] \sqsubset [t'_1, ..., t'_q] \Leftrightarrow q > p$ and $\forall t_i \exists j : t_i = t'_j$
- private $\prec$ *default* $\prec$ protected $\prec$ public

Next, separately for each version v' and its predecessor version v we apply our light-weight parser on both v and v' to get the set $M = M_v$ which contains the methods in the predecessor version, and $M' = M_{v'}$ which contains the methods in the newer version. Then, we look for entities $e' \in M'$ and $e \in M$ which satisfy the conditions[1] in Table 1 and thus indicate that the corresponding method *may have been refactored*. In this case we call the tuple (v', e', v, e) a refactoring candidate, i.e., entity e' in version v' possibly results by refactoring from entity e in version v.

So far, we have not taken into account changes to the body of a method. We have two extreme options here. First, we could require that no changes to the body are allowed. In this case, our signature-based analysis would only find few refactorings and this strong restriction would not even be sufficient to preserve the external behavior, e.g., due to method overloading. On the other hand we could simply ignore changes to the body of the method. In this case, we would get too many refactoring candidates. Thus we need a better way to take changes to the method body into account and in Section 2.4 we will use clone detection to this end.

[1] For some of the comparisons, not all elements of the tuples are relevant. We use the value "*" in the tuple to indicate that the value of this part of the tuple does not matter.

Table 2. Conditions for Structural Refactorings

Move Method	$\exists\,(c, m, p, r, *) \in M_R$ and $\exists\,(c', m, p, r, *) \in M_A$
Move Field	$\exists\,(c, f, t) \in F_R$ and $\exists\,(c', f, t) \in F_A$
Move Class	$\exists\,(p, n) \in C_R$ and $\exists\,(p', n) \in C_A$
Move Interface	$\exists\,(p, n) \in I_R$ and $\exists\,(p', n) \in I_A$
Rename Class	$\exists\,(p, n) \in C_R$ and $\exists\,(p, m) \in C_A$ where $n \neq m$

M_R resp. M_A is the set of all methods removed resp. added in the current transaction. F_R resp. F_A is the set of all fields removed resp. added in the current transaction. C_R resp. C_A is the set of all classes removed resp. added in the current transaction. I_R resp. I_A is the set of all interfaces removed resp. added in the current transaction.

Structural refactorings

In contrast to local refactorings, structural refactorings can involve multiple files because entities may have been moved around between files. Thus, it is no longer sufficient to only compare entities of a version of a file and its predecessor version. Instead, for each transaction t we consider all versions $v'_1, \ldots, v'_m$ of JAVA files in that transaction and compute the set C' of all classes in this transaction as the union $C' = C_{v'_1} \cup \cdots \cup C_{v'_m}$. After that, we identify the predecessor versions $v_1, \ldots, v_m$ for the versions in t, such that v_i is the predecessor of v'_i. Note, that the predecessors of two different versions in t are usually not checked in as part of the same transaction. The set of all predecessor classes is $C = C_{v_1} \cup \cdots \cup C_{v_m}$. From these two sets we can now compute the set of added classes $C_A = C' - C$ and the set of removed classes $C_R = C - C'$. Analogously we compute the sets I_A, I_R, M_A, M_R, F_A, and F_R.

Finally, we compare these sets as described in Table 2 to identify which entities are candidates for refactorings: Let e be the predecessor entity of the entity e' and v' and v are the versions containing these entities. If e' and e satisfy one of the conditions in Table 2, then the tuple (v', e', v, e) is a refactoring candidate.

Thus, for each transaction t we get a set RC_t of refactoring candidates that have been detected in this transaction and for the whole software archive the set $RC = \bigcup_t RC_t$ that contains all release candidates found in the transactions of the software archive.

2.4. Ranking the Refactoring Candidates

The analysis step described in the previous section yields candidates for refactorings. As it is in general undecidable, whether a change preserves the external behavior of the program or not, we need an approach to *estimate* how good a refactoring candidate is. We rank the candidates depending on the results of a test for equality and of a code-clone detection algorithm.

In the following we assume that the function $\mathsf{body} : E \times V \rightarrow \mathsf{String}$ yields the body of an entity in a certain version, i.e., the code fragment contained in that entity. For entities that do not have a body, for example entities of type field, the function yields an empty string.

Code-clone detection

For each refactoring candidate (v', e', v, e) we compare $b' = \mathsf{body}(e', v')$ and $b = \mathsf{body}(e, v)$. If they are equal, the body of the refactored entity has not been changed.

If the bodies b' and b are not equal, we use code-clone detection to determine whether the bodies are similar in a way that it is likely that the external behavior did not change.

Currently, we use the tool CCFINDER [12, 16] for code-clone detection: for each refactoring candidate (v', e', v, e) we run CCFINDER to compare $b' = \mathsf{body}(e', v')$ and $b = \mathsf{body}(e, v)$.

For two JAVA code fragments j_1, j_2 we define that j_2 is a code-clone of j_1, short $j_1 \rightharpoonup j_2$, if j_1 is equal to j_2 or if j_1 can be transformed into j_2 by only performing the following operations (to achieve the described behavior we call CCFINDER with the following JAVA-language options: -rabcd-fr-kmnop-r-s-u-v):

Adding or deleting white spaces. Generally, a sequence of white spaces is interpreted as one single white space by the clone detection because the number of white spaces has no effect on the behavior of the program code.

Adding/deleting/changing comments. As comments do not have any effect on the execution of a JAVA program, they are ignored in the clone detection.

Changing visibility. Although changing the visibility of a method or a field may change external behavior, we ignore visibility keywords in the clone detection. This is meaningful when we consider candidates for refactorings of kinds *move class*, *rename class*, and *move interface* because in the same transaction within the affected class/interface *hide/unhide method* refactorings could have taken place.

Adding/removing package name. In JAVA references to classes can be written fully qualified, i.e. including the package, or short by the class name (if the referenced class is visible or imported). Adding or removing the leading package name to/from class references, thus, does not change the functionality. However, as we do not test if the affected class is imported or anyway visible, it may happen that a class with the same name but in another package is referenced after the change.

Consistently renaming variables. Consistently renaming local variables does not change functionality of the code. However we also accept consistent renaming of variables that are defined somewhere else. This can cause us to identify code fragments as code-clones that have different external behavior, but it is nevertheless useful because in the same transaction another refactoring could have taken place that has renamed the affected variables at the location where they are defined.

Consistently renaming method names. For candidates of refactorings of kinds *move class*, *rename class*, and *move interface* this allows us to find code-clones even if methods in the affected class/interface have been renamed in the same transaction. However, there is a chance that we identify code-clones for code fragments with different external behavior.

Consistently renaming references to member names. Members (e.g., fields or methods) of classes are called in JAVA using the syntax `classname.membername`. We identify two code fragments as clones even if in these calls member names are consistently renamed. This approach has one big advantage: If we look at a candidate for a refactoring, we can identify the newer version of the affected entity as a code-clone even if in the same transaction other methods or fields that are called within the refactoring candidate have been renamed.

Consistently renaming types. When a type (e.g., a class or an interface) has been renamed, all references to this class have to be renamed, too. Thus, allowing consistent renaming of types allows us to find code-clones between older and newer versions of bodies for refactoring candidates even if in the same transaction refactorings of kind *rename class* have occurred for classes that are referenced in the body of the refactoring candidate under view.

Code similarity categories

Depending on the results of the equality test and of the code-clone detection, we classify each refactoring candidate (v', e', v, e) to belong into one of the following *code similarity* categories: Let $\mathsf{codesim} : RC \rightarrow \{\text{EQUAL}, \text{CLONE}, \text{NOCLONE}\}$ be a function that yields the following values for a refactoring candidate (v', e', v, e):

Code Similarity: EQUAL**.** The bodies of both entities are equal, thus $\mathsf{body}(e', v') = \mathsf{body}(e, v)$. This means the body of the possibly refactored entity has not been changed by the transformation from e to e'.

Code Similarity: CLONE**.** The bodies of both entities are clones: $\mathsf{body}(e', v') \rightharpoonup \mathsf{body}(e, v)$, but $\mathsf{body}(e', v') \neq \mathsf{body}(e, v)$.

Code Similarity: NOCLONE**.** The bodies of both entities are neither equal nor clones of each other.

The above categories can be ordered according to the risk or likelihood that the contained refactoring candidates are no refactorings, but change the external behavior. EQUAL has the smallest risk, CLONE still has a low risk, whereas NOCLONE has a high risk. Thus, the code similarity helps to identify refactoring candidates that probably are no refactorings. For the evaluation (see Section 3) we chose two category sets: $LOW = \{\text{EQUAL}, \text{CLONE}\}$ and $HIGH = \{\text{NOCLONE}\}$.

3. Evaluation

In the previous section we have presented a technique to find candidates for refactorings and described how code-cloning helps to rank these candidates. In this section we present an evaluation of our technique concerning two aspects — its recall and precision:

- How many of the overall refactorings in a project are automatically detected?
- How correct are the computed refactoring candidates?

Let RC be the set of refactoring candidates found, DR be the set of documented refactorings, and R be the set of all refactorings in the archive. Then the recall is defined as the number of correct answers found divided by all correct answers, i.e., $\mathsf{recall} = \frac{|RC \cap R|}{|R|}$, and the precision is the number of correctly found answers divided by all found answers, i.e., $\mathsf{precision} = \frac{|RC \cap R|}{|RC|}$. Unfortunately, with manual inspection it would probably take months to find all undocumented refactorings $R - DR$, and it would require expert knowledge of all modules of the inspected software project. So, we actually look at two cases:

Documented refactorings. For documented refactorings we compute the recall as $\mathsf{drecall} = \frac{|RC \cap DR|}{|DR|}$, i.e., how many of the documented refactorings do we find. Computing the precision $\mathsf{dprecision} = \frac{|RC \cap DR|}{|RC|}$ does not provide much insight, as the set RC also contains candidates for undocumented refactorings.

Undocumented refactorings. For undocumented refactorings we use random sampling to estimate the precision. For the small number of samples, we manually check the correctness. In the sequel we will denote the estimated precision by $\widetilde{\mathsf{precision}}$.

3.1. Recall: Documented Refactorings

Evaluation Setup To evaluate how many of the actually performed refactorings our technique can detect, we manually inspected the log messages of two open-source projects,

CVS Module	jakarta-tomcat	jEdit
First transaction	1999-10-08	2001-09-02
Last transaction	2004-11-21	2004-11-01
# Transactions	5096	2141
# Committers	40	6

Table 3. Key data for the evaluated projects

JEDIT and TOMCAT, and searched for *documented refactorings* in these. The key data for these projects is shown in Table 3. For JEDIT we considered the CVS module `jEdit` which contains the complete version history for this project. For TOMCAT we have analyzed the module `jakarta-tomcat` which contains the version history for the TOMCAT 3.x series. The latest production quality release was 3.3 in April 2002, since then only bug fixes have been made in this module.

The manual extraction of the documented refactorings works as follows: If the developer who has checked-in a set of new versions into the repository, has described her changes as being a refactoring in the log message and we have been able to manually identify the actual refactorings in the code, we consider these refactorings as *documented*.

Manual inspection is very tedious, as the descriptions in the log messages are typically very vague, e.g., the developer just writes that she has "done a refactoring", or she is a bit more precise and states the kind of refactoring, e.g., "moved classes".

In both cases, it is not explicit which entities have been refactored and *how* exactly. For example, we know that some classes have been moved, but we do not know exactly which classes, neither do we know the exact source package and the exact target package. This means, that we have to examine the differences between the previous and the checked-in version of the source code to identify which refactoring has actually been documented here.

Sometimes log messages are misleading. For example, 9 log messages in the TOMCAT archive indicated refactorings of kind *move class*, 2 of kind *rename class*, and 9 of kind *move method*, but in all these cases it turned out that the programmers created new classes or methods, but did not delete the old ones. Another misleading log message in TOMCAT starts with the words "moved code out of facade into the real classes". One could think that this log message indicates several *move method* refactorings, but in fact code has been moved, but the old methods have been preserved to delegate method calls to the new implementation.

The result of the manual inspection for the transactions $t_1, \ldots, t_n$ is a set of *documented refactorings* for each of the transactions: $DR_{t_1}, \ldots, DR_{t_n}$. And the set of all documented refactorings is simply $DR = \bigcup_{t_i} DR_{t_i}$.

For each documented refactoring $(v', e', v, e,) \in DR_{t_i}$ we look if we have found this refactoring also as a refactoring candidate in our analysis, i.e., if $(v', e', v, e) \in RC_{t_i}$ and, in case, what its code similarity category is. This gives us an overview of which documented refactorings have been detected and how they were ranked. The results are shown in Tables 4 and 5 and are discussed in more detail below.

Results for TOMCAT and JEDIT: Table 4 shows the results of our evaluation for the project TOMCAT and Table 5 for JEDIT, each broken down by the kinds of refactoring, e.g. RC_k is the set of refactoring candidates found of kind k. The first number in each line is the absolute number of refactoring candidates of that kind found in the project. The next three columns are breakdowns of these numbers for each code similarity category. Columns 6 to 10 contain the number of documented refactorings, the number of how many of those have been actually found broken down to the code similarity categories (Columns 7-9), and the overall number of found documented refactorings. The right-most two columns contain the percentage of found documented refactorings using all code similarity categories (Column 11) respectively using only the code similarity categories EQUAL and CLONE (Column 12).

In the log messages of TOMCAT 222 refactorings[2] have been documented. We found 206 of these. In other words, we found 93% of the documented refactorings. For JEDIT, less refactorings have been documented, exactly 37, from which we found 32 or 86%.

The evaluation shows the highest recall for structural refactorings, here we found 97% of all documented refactorings for TOMCAT and 92% for JEDIT. Note, that for the latter no refactorings of the kinds *move interface* and *move field* have been documented.

The results for local refactoring are also good: For TOMCAT we found more than half and for JEDIT even 83% of the documented local refactorings. Only few *hide/unhide method* refactorings have been documented. We found the one *hide method* refactoring from the TOMCAT logs and both *unhide methods* from the JEDIT logs. *Rename method* seems to be the most problematic refactoring kind for our current technique: None of the two documented *rename method* refactorings in JEDIT and only slightly more than half for TOMCAT have been detected.

While for local refactorings code-clone detection does not increase the recall, it seems to be quite useful for structural refactorings: for TOMCAT 10% and for JEDIT 17% of all detected documented structural refactorings have a code similarity of CLONE.

In total, the recall of our refactoring reconstruction is 93% for the documented refactorings in TOMCAT respectively 86% for JEDIT. Taking refactoring candidates with a high risk into account was especially useful for JEDIT:

[2]There have been more documented refactorings, but we only consider here the kinds of refactorings currently covered by our analysis.

Refactoring Kind	all				documented						
	$\|RC_k\|$	$\|RC_k^{\text{EQUAL}}\|$	$\|RC_k^{\text{CLONE}}\|$	$\|RC_k^{\text{NOCLONE}}\|$	$\|DR_k\|$	$\|RC_k^{\text{EQUAL}} \cap DR_k\|$	$\|RC_k^{\text{CLONE}} \cap DR_k\|$	$\|RC_k^{\text{NOCLONE}} \cap DR_k\|$	$\|RC_k \cap DR_k\|$	drecall$_k$	drecall$_k^{LOW}$
Local Refactorings											
Rename Method	104	48	32	24	13	7	0	0	7	54%	54%
Hide Method	42	21	4	17	1	0	0	1	1	100%	0%
Unhide Method	106	82	3	21	0	0	0	0	0	–	–
Add Parameter	136	30	6	100	7	3	0	0	3	43%	43%
Remove Parameter	34	3	2	29	3	3	0	0	3	100%	100%
Total	422	184	47	191	24	13	0	1	14	**58%**	**54%**
Structural Refactorings											
Move Class	540	100	12	428	37	19	4	14	37	100%	62%
Rename Class	511	6	16	489	4	1	2	1	4	100%	75%
Move Interface	21	21	0	0	21	21	0	0	21	100%	100%
Move Field	1915	–	–	–	46	45	0	0	45	98%	98%
Move Method	3283	2112	199	972	90	53	14	18	85	94%	74%
Total	6270	2239	227	1889	198	139	20	33	192	**97%**	**80%**
Overall Total	6692	2423	274	2080	222	152	20	34	206	**93%**	**77%**

Table 4. Refactoring candidates found in the TOMCAT archive

Without these candidates the recall decreases to about 38% (14 out of 37 documented refactorings found). For TOMCAT, however, omitting high-risk candidates causes only to lose 34 refactorings. Thus, the recall is still 77% without these high-risk candidates.

An interesting question is, why some of the documented refactorings have not been found by our analysis. Thus, we have looked in the source code again for each documented refactoring that has not been found. It turned out that many of these refactorings were not found for the same reason—the application of *multiple refactorings* to the same software artifact. For example, in Transaction 1269 in the TOMCAT archive the method `processServlets()`, which has one parameter of type `Enumeration` has been moved from the class `org.apache.tomcat.core.Context` to `org.apache.tomcat. context.WebXmlInterceptor`. In the same change, a second parameter of type `Context` has been added to the method definition. According of the definition of entities this has the following consequences: The entity (c, m, p, r, v) has been changed to (c', m, p', r, v) with $c \neq c' \wedge p \neq p'$. So, according to Tables 1 and 2, this change is neither a candidate for *move method*, nor for *add parameter*. Thus, multiple refactorings performed on the same entity are a major problem for our analysis technique. While no multiple refactorings have been documented for JEDIT, in TOMCAT we found out that 13 documented refactorings (8 of kind *move method*, 3 of kind *add parameter*, 2 of kind *rename method*) have not been found because of this problem. In other words, multiple refactorings are responsible for most of the cases, where we did not find the refactoring. One approach to solve this problem is to add new conditions to the signature-based analysis for frequent combinations of refactorings. For example, the combination of *move class* and *rename class* can be captured if we add the following condition to Table 2: $\exists\,(p, n) \in C_R$ and $\exists\,(p', m) \in C_A$ where $n \neq m \wedge p \neq p'$

3.2. Precision: Undocumented Refactorings

As our analysis will also yield correct refactoring candidates, that are just not documented, computing the precision for documented refactorings does not help. Instead, we would like to know how many of all refactoring candidates we found are really refactorings. As there are for example 6692 refactoring candidates in the TOMCAT archive and only 206 of these can be automatically matched with documented refactorings, we would have to manually check the remaining 6486 candidates. Instead we chose a sampling approach to estimate the precision.

Evaluation Setup In this evaluation, we took the set $RC - DR$ of all refactoring candidates found by our analy-

Refactoring Kind	all				documented						
	$\|RC_k\|$	$\|RC_k^{\text{EQUAL}}\|$	$\|RC_k^{\text{CLONE}}\|$	$\|RC_k^{\text{NOCLONE}}\|$	$\|DR_k\|$	$\|RC_k^{\text{EQUAL}} \cap DR_k\|$	$\|RC_k^{\text{CLONE}} \cap DR_k\|$	$\|RC_k^{\text{NOCLONE}} \cap DR_k\|$	$\|RC_k \cap DR_k\|$	drecall_k	drecall_k^{LOW}
Local Refactorings											
Rename Method	46	15	19	12	2	0	0	0	0	0%	0%
Hide Method	42	24	1	17	0	0	0	0	0	–	–
Unhide Method	41	29	0	12	2	2	0	0	2	100%	100%
Add Parameter	260	24	1	235	3	0	0	3	3	100%	0%
Remove Parameter	84	12	1	71	5	2	0	3	5	100%	40%
Total	473	104	22	347	12	4	0	6	10	**83%**	**33%**
Structural Refactorings											
Move Class	226	7	1	218	8	3	1	4	8	100%	50%
Rename Class	862	18	7	837	2	1	1	0	2	100%	100%
Move Interface	3	1	0	2	0	0	0	0	0	–	–
Move Field	373	–	–	–	0	–	–	–	0	–	–
Move Method	382	188	53	141	15	2	2	9	13	87%	27%
Total	1846	214	61	1198	25	6	4	13	23	**92%**	**40%**
Overall Total	2319	318	83	1545	37	10	4	19	32	**86%**	**38%**

Table 5. Refactoring candidates found in the JEDIT archive

sis in the TOMCAT archive minus the candidates found for documented refactorings. From this set we randomly chose a number of candidates and manually checked whether they are actual refactorings. Once we realized that the precision depends on the code similarity category, we decided to actually perform two different samplings. One where we only considered candidates of categories EQUAL and CLONE (category set LOW), i.e. where there have been only modest changes in the body of the code, and one considering only those of the category NOCLONE (category set $HIGH$), i.e., with more differences and a higher risk, that the candidates are no actual refactorings. More precisely, a sample set $S_{k,s}^M \subseteq RC_k$ of size s contains only candidates of kind k which belong to one of the categories in M, i.e., $|S_{k,s}^M| = s$ and $\forall r \in S_{k,s}^M : \mathsf{codesim}(r) \in M$. For the evaluation we chose a sample size $s = 5$.

For each of these refactoring candidates we manually look in the source code, if the described refactoring has actually taken place. Thus, we find out how precise our candidates are for the particular refactoring kinds and how the precision is influenced by the code similarity of the refactoring candidate.

Results for TOMCAT: Table 6 shows the results of our sampling approach applied to the TOMCAT archive. For refactorings of kind *Move Interface* we did not find any undocumented candidates, and for *Remove Parameter* only two undocumented candidates to take samples.

For estimating the average precision on all refactoring kinds from the estimated precisions for each refactoring kind we have to take the ratio of candidates for each refactoring kind compared to the number of all refactoring candidates into account, thus we get

$$\widetilde{\mathsf{precision}}^{LOW} = \frac{1}{|RC - DR|} \sum_{k \in K} \widetilde{\mathsf{precision}}_k^{LOW} |RC_k - DC_k|$$

where K is the set of all refactoring kinds. To estimate the precision $\widetilde{\mathsf{precision}}_{local}^{LOW}$ for local refactorings we use the set K_{local} of local refactorings instead of K. Analogously we estimate $\widetilde{\mathsf{precision}}_{struct}^{LOW}$. As we take separate samples for low-risk and high-risk candidates, we additionally compute $\widetilde{\mathsf{precision}}_{struct}^{HIGH}$ and $\widetilde{\mathsf{precision}}_{local}^{HIGH}$ analogously.

Using these formulas, for similarity categories EQUAL and CLONE we get a precision of 92% for local refactorings and of 72% for structural refactorings. In total, about 73% of all low-risk refactoring candidates are correct.

Taking candidates with a high risk into account seems to be only meaningful for local refactorings: nearly half of the high-risk candidates for local refactorings in our sample turned out to be correct. However, for structural refactor-

Refactoring Kind	$\lvert S^{LOW}_{k,5} \cap R \rvert$	$\widetilde{\text{precision}}^{LOW}_{k,5}$	$\lvert S^{HIGH}_{k,5} \cap R \rvert$	$\widetilde{\text{precision}}^{HIGH}_{k,5}$
Local Refactorings				
Rename Method	5	100%	0	0%
Hide Method	5	100%	3	60%
Unhide Method	4	80%	3	60%
Add Parameter	5	100%	2	40%
Remove Parameter	2	100%	5	100%
Average		**92%**		**48%**
Structural Refactorings				
Move Class	5	100%	0	0%
Move Interface	–		–	
Move Field	3	60%	–	
Rename Class	4	80%	0	0%
Move Method	4	80%	0	0%
Average		**72%**		**0%**
Overall Average		**73%**		**4%**

Table 6. Precision of the refactoring candidates in the random samples (TOMCAT only)

ings none of the detected candidates with code similarity NOCLONE were correct.

3.3. Estimating the number of refactorings

As mentioned before, for a project like TOMCAT the total number of refactorings can not be exactly determined with acceptable effort. On the other hand, this number could be very interesting, as it tells us how many refactorings of the kinds covered by our analysis have actually occurred. Nowadays there exist tools to support all of these kinds of refactorings. If we know that the number of refactorings in a project is high, the effort to introduce such a tool will soon pay off.

Assuming that the recall for undocumented refactorings would be the same as for documented ones, we can make a very rough guess, on how many refactorings have actually been performed in the archive, i.e, $|R|$. From the definitions of precision and recall, we immediately get the formula to compute $|R| = \frac{\text{precision} \cdot |RC|}{\text{recall}}$. Using $\widetilde{\text{precision}}^{LOW}$ instead of precision and drecall^{LOW} instead of recall we approximate the total number of refactorings in the TOMCAT archive using $|R| = \frac{73\% \cdot 6692}{57\%} \approx 8570$. Admittedly, this calculation is statistically not valid, because of the small sample size and the assumption that documented and undocumented refactorings have the same statistical distribution. But at least, we think, that is a good guess and that the real number is between 7000 and 10000. Before the evaluation, when we asked ourselves, what the real value of $|R|$ might be, we didn't have a clue whether it was in the order of hundreds, thousands, or ten thousands.

3.4. Threats to Validity

We have studied two open source projects. Although they are very different, we cannot claim that their version histories would be representative for all kinds of software projects — in particular closed-source projects.

Although we have very carefully inspected the source code to determine the documented refactorings, we are not experts involved in the development of one of these systems, and may have misclassified some of the changes.

To decide if a change is really a refactoring, i.e., it just changes the code structure but not the code behavior, it is not sufficient to look at the signature change of the maybe refactored entity and the code similarity. E.g., if new parameters are added to a method, these parameters have to be used in the respective method calls. But they have to be set to a value that does not change the behavior. Although our technique does not test this, nor use any other semantic information, its recall and precision are surprisingly high.

Finally, for the estimation of the precision we used 10 samples per kind of refactoring (5 for low and 5 for high risk categories). This sample size may be too small to yield statistically significant results.

4. Related Work

One of the first publications about refactorings was William Opdyke's PhD thesis [13] which contains general information on refactoring and many examples for various refactoring kinds. Fowler's book [7] presents a large refactoring catalog which also includes the refactorings identified by our analysis.

The differencing approach that we use to detect changed code blocks is very similar to the technique used by Apiwattanapong's JDIFF tool [2] when operating on method-level (it can also work on node-level, which is not useful for our current analysis). In contrast, Horwitz [11] and Raghavan [14] represent versions as graphs that take semantic information like control and data flow into account, and use graph differencing algorithms to identify the changes between different versions of the program.

Demeyer et al. [4] looked for changes in software metrics like method size, class size, or number of inherited or overwritten methods to detect refactorings. The precision of their analysis seems to be rather low, for example, for move method refactorings (restricted to sub, super and sibling classes) the average precision over all projects they evaluated is 23% (see Table 5 in [4]).

Recently, Dig et al. [5] presented a method that is very similar to our signature-based analysis. While we use the transactions to prevent us from comparing whole versions to find initial candidates for refactorings, they apply a hashing technique to identify similar methods. As they did not provide any evaluation results, we cannot make a quantitative comparison to our approach.

Antoniol et al. [1] used a vector space technique to compare identifiers in different classes to detect renaming and splitting of classes. However the approach does not perform very well, if many changes have been performed within the classes.

Van Rysselberghe and Demeyer [15] use a visualization technique called dotplots to identify possible refactorings. Each dot indicates that two lines of code match. Moved code is visible as diagonal lines in this visualization. Unfortunately, the paper does not contain any evaluation results.

5. Conclusions

Refactoring reconstruction is a prerequisite for many applications. In this paper we presented a technique that combines signature-based analysis to detect and clone-detection to rank refactoring candidates. As expected the evaluation shows that we can trade recall for precision by choosing high-risk or low-risk similarity categories. For TOMCAT using low-risk categories we got a recall of 77% and a precision of 73% for all refactoring kinds. Considering only structural refactorings recall and precision increased even to 80% respectively 92%.

In our future work we want to investigate some of those applications briefly discussed in the introduction, in particular applying refactoring reconstruction to relate refactorings to other changes and to software metrics. We also intend to improve the technique, experiment with other code-clone detection methods, and cover more kinds of refactorings.

Acknowledgments Michael Burch gave helpful comments on a draft of this paper. Carsten Görg was involved in the development of the early versions of our analysis. Johanna Vomfei helped with the manual inspection of the documented refactorings.

References

[1] G. Antoniol, M. D. Penta, and E. Merlo. An automatic approach to identify class evolution discontinuities. In *Proc. International Workshop on Principles of Software Evolution (IWPSE 2004)*.

[2] T. Apiwattanapong, A. Orso, and M. J. Harrold. A differencing algorithm for object-oriented programs. In *Proc. International Conference on Automated Software Engineering (ASE 2004)*.

[3] P. Cederqvist. *Version Management with CVS*, 2003. http://www.cvshome.org/docs/manual/.

[4] S. Demeyer, S. Ducasse, and O. Nierstrasz. Finding refactorings via change metrics. In *Proc. Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA 2000)*.

[5] D. Dig, C. Comertoglu, D. Marinov, and R. Johnson. Automatic detection of refactorings for libraries and frameworks. In *Proc. of ECOOP Workshop on Object-Oriented Reengineering (WOOR 2005)*.

[6] M. Fischer, M. Pinzger, and H. Gall. Populating a release history database from version control and bug tracking systems. In *Proc. International Conference on Software Maintenance (ICSM 2003)*.

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

[8] C. Görg and P. Weißgerber. Detecting and visualizing refactorings from software archives. In *Proc. International Workshop on Program Comprehension (IWPC 2005)*.

[9] C. Görg and P. Weißgerber. Error Detection by Refactoring Reconstruction. In *Proc. International Workshop on Mining Software Repositories (MSR 2005)*.

[10] J. Henkel and A. Diwan. Catchup!: capturing and replaying refactorings to support api evolution. In *Proc. International Conference on Software Engineering (ICSE 2005)*.

[11] S. Horwitz. Identifying the semantic and textual differences between two versions of a program. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1990.

[12] T. Kamiya, S. Kusumoto, and K. Inoue. Ccfinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions On Software Engineering*, 28(7), 2002.

[13] W. F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois, 1992.

[14] S. Raghavan, R. Rohana, D. Leon, A. Podgurski, and V. Augustine. Dex: A semantic-graph differencing tool for studying changes in large code bases. In *Proc. International Conference on Software Maintenance (ICSM 2004)*.

[15] F. V. Rysselberghe and S. Demeyer. Reconstruction of successful software evolution using clone detection. In *Proc. International Workshop on Principles of Software Evolution (IWPSE 2003)*.

[16] Y. Ueda, Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue. Gemini: Code clone analysis tool. In *Proc. International Symposium on Empirical Software Engineering (ISESE 2002)*.

[17] P. Weißgerber and S. Diehl. Are Refactorings less error-prone than other changes? In *Proc. International Workshop on Mining Software Repositories (MSR 2006)*.

[18] T. Zimmermann and P. Weißgerber. Preprocessing CVS data for fine-grained analysis. In *Proc. International Workshop on Mining Software Repositories (MSR 2004)*.

[19] T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. In *Proc. International Conference on Software Engineering (ICSE 2004)*.

[20] L. Zou and M. W. Godfrey. Detecting merging and splitting using origin analysis. In *Proc. Working Conference on Reverse Engineering (WCRE 2003)*.