Comparison of Similarity Metrics for Refactoring Detection

Benjamin Biegel
University of Trier,Germany
biegel@uni-trier.de

Quinten David Soetens*
University of Antwerp,Belgium
quinten.soetens@ua.ac.be

Willi Hornig
University of Trier,Germany
s4wihorn@uni-trier.de

Stephan Diehl
University of Trier,Germany
diehl@uni-trier.de

Serge Demeyer
University of Antwerp,Belgium
serge.demeyer@ua.ac.be

ABSTRACT

Identifying refactorings in software archives has been an active research topic in the last decade, mainly because it is a prerequisite for various software evolution analyses (e.g., error detection, capturing intent of change, capturing and replaying changes, and relating refactorings and software metrics). Many of these techniques rely on similarity measures to identify structurally equivalent code, however, up until now the effect of this similarity measure on the performance of the refactoring identification algorithm is largely unexplored. In this paper we replicate a well-known experiment from Weißgerber and Diehl, plugging in three different similarity measures (text-based, AST-based, token-based). We look at the overlap of the results obtained by the different metrics, and we compare the results using recall and the computation time. We conclude that the different result sets have a large overlap and that the three metrics perform with a comparable quality.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Restructuring, reverse engineering, and reengineering*; D.2.8 [**Software Engineering**]: Metrics

General Terms

Algorithms, Measurement

Keywords

Refactoring, replication experiment, similarity metrics, code clones, mining software repositories, software evolution

*This work has been carried out in the context of the Interuniversity Attraction Poles Programme - Belgian State – Belgian Science Policy, project *MoVES*.

MSR'11, May 21–22, 2011, Waikiki, Honolulu, HI, USA

1. INTRODUCTION

Refactoring has been widely accepted as one of the principal techniques to restructure a software system's design and decrease it's complexity. In his book on refactoring [7], Fowler defines refactoring as *"a change to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior."*

The key idea is to redistribute instance variables and methods across the class hierarchy in order to simplify the structure of a software system whilst preserving the behavior of the system and consequently prepare the software for future extensions [12]. If applied well, refactoring is said to improve the design of software, make software easier to understand, help to find bugs, and help to program faster [7]. 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 (SWEBOK) [1].

Identifying refactorings in software repositories is a prerequisite for various applications. It can be used for error detection [8] or as a reverse engineering technique to find the intent of certain changes [19, 21]. It can also be used to capture the performed refactorings and replay them on other software that depends on the refactored system [9]. Or it can be used to correlate refactorings to software metrics and as such provide new insights on how software projects evolve over time [18].

For these reasons refactoring detection has been an active research topic in the academic community. Weißgerber and Diehl proposed a signature-based technique that uses similarity metrics to rank refactoring candidates [22].

In this paper we try to replicate their results and investigate how it compares when we use other similarity metrics. We use their algorithm and rank the refactoring candidates based on three different similarity metrics: shingles, CCFinder and JCCD. shingles is a text-based technique taken from natural language processing to gauge the similarity between two documents. CCFinder and JCCD are both code clone detection tools. The fundamental difference is that JCCD is an AST-based clone detector whereas CCFinder is token-based. In this paper we consider two main research questions: (i) How big is the influence of a similarity metric within the signature-based refactoring detection? and (ii) How do the similarity metrics differ?

The rest of this paper is structured as follows. In Section 2 we summarize the algorithm used to detect refactorings and elaborate on the different similarity metrics used

in Section 3. We then look for an answer for our two research questions in Section 4 before ending with a summary of some threats to the validity of our experiment (Section 5), the related work (Section 6) and the conclusions (Section 7).

2. SIGNATURE-BASED REFACTORING DETECTION

There exist several refactoring detection techniques. One of which is the signature-based technique by Weißgerber and Diehl which has already been proven to be a good approach and has also demonstrated success on many applications [22]. Therefore we chose this technique to evaluate the effect of choosing different similarity metrics on the performance and accuracy of the refactoring detection.

Weißgerber's algorithm starts with a preprocessing step that stores the most important data of the source code repository in a relational database in order to get fast access to it. Then the technique looks for added, changed, or removed entities (classes, interfaces, fields, or methods) to obtain refactoring candidates. Most important for this paper, the technique relies on a similarity metric to rank these candidates to indicate which are more likely to retain their behavior and are thus more likely to be real refactorings.

The preprocessing step is used to extract and clean up the data about the versioning of the software system. This results in versions and transactions. A *version* describes one revision of a file in the software repository. A *transaction* is the set of versions that were committed to the software repository at the same time by the same developer. A detailed description of how data from a software repository can be preprocessed can be found in [25].

After the preprocessing a light-weight parser computes for each version $v \in V$ of a JAVA file the following sets:

- C_v: The set of classes of the software system. This set contains elements (p, n, w), where p is the name of the package to which the class belongs, n is the class name and w is the visibility of the class.
- I_v: The set of interfaces of the software system. This set contains elements (p, n, w), where p is the name of the package to which the interface belongs, n is the interface name and w is the visibility of the interface.
- M_v: The set of methods of the software system. This set contains tuples (c, m, p, r, w) where c is the fully-qualified name of the class to which the method belongs, m is the method name, p is the parameter list of the method, r is the return type of the method and w is the visibility of the method.
- F_v: The set of fields of the software system. This set contains elements (c, f, t, w), where c is the fully-qualified name of the class to which the field belongs, f is the field name, t is the field type and w is the visibility of the field.

The next step in the technique is to determine which entities were added and removed in each transaction. This results in the sets C_t^+, C_t^-, I_t^+, I_t^-, M_t^+, M_t^-, F_t^+ and F_t^-. C_t^+ is the set containing the classes that were added in transaction t, C_t^- is the set that contains the classes that were removed in transaction t. The other sets contain the added and removed interfaces, the added and removed methods, and the added and removed fields.

Table 1: Conditions for refactoring candidates

Refactoring Kind	Condition
MoveClass	$\exists(p, n, w) \in C_t^-$ and $\exists(p', n, w) \in C_t^+$
RenameInterface	$\exists(p, n, w) \in I_t^-$ and $\exists(p, n', w) \in I_t^+$
HideField	$\exists(c, f, t, w) \in F_t^-$ and $\exists(c, f, t, w') \in F_t^+$ and $w' \prec w$ $\exists(c, m, p, r, w') \in F_t^+$ and $w' \prec w$
UnhideMethod	$\exists(c, m, p, r, w) \in F_t^-$ and $\exists(c, m, p, r, w') \in F_t^+$ and $w \prec w'$
AddParameter	$\exists(c, m, p, r, w) \in M_t^-$ and $\exists(c, m, p', r, w) \in M_t^+$ and $p \sqsubset p'$
RemoveParameter	$\exists(c, m, p, r, w) \in M_t^-$ and $\exists(c, m, p', r, w) \in M_t^+$ and $p \sqsupset p'$

The order $\sqsubset$ on lists of types and the order $\prec$ on visibility levels is defined 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

Using these sets and the tuples they contain, we can formally describe the criteria used in the signature-based analysis to find the refactoring candidates. The sets of added and removed entities are compared as described in Table 1 to identify which entities are candidates for refactorings[1]. Let s' be a signature of a removed entity and s be a signature of an added entity and k a refactoring kind. If s' and s satisfy one of the conditions in Table 1 for the particular refactoring kind k, then the tuple $(s, k, s')_t$ is a candidate for a refactoring of the kind k in the corresponding transaction t. This way we can find a set of refactoring candidates RC_t for each transaction t.

2.1 Detecting Multiple Refactorings

A common problem in the field of refactoring detection is when more than one refactoring is performed on the same artifact. For instance when a RenameMethod and an AddParameter refactoring are performed on the same method, then the conditions in Table 1 will not generate a candidate. To counteract this effect, the refactoring detection technique is extended with the possibility to capture multiple refactorings. This is done by adapting the conditions for refactoring candidates in such a way that the only thing that matters is the value of the element in the tuple that is supposed to be changed. This is done by changing the conditions in Table 1 and changing the values of the elements in the tuples that do not matter by a wildcard "*". For example, a candidate for an AddParameter refactoring is created if $\exists(*, *, p, *, *) \in M_t^-$ and $\exists(*, *, p', *, *) \in M_t^+$ and $p \sqsubset p'$.

This new type of candidates is called *weak candidates* to distinguish such candidates from the ones described in Table 1, which are called *strong candidates* from now on.

While weak candidates allow to identify refactorings for an entity even if multiple refactorings have been applied to that entity, they have an important disadvantage: As the conditions for such candidates are much weaker than the conditions for strong candidates, the chance that a weak

[1]We only show a subset of the conditions to identify refactorings, the other Rename, Hide, Unhide and Move refactoring conditions can be defined analogously.

candidate describes a real refactoring is much lower. This also means that taking weak candidates into account will result in a large number of candidates that are likely to be wrong. However, this problem is not as severe as it seems because the technique is at this point only working with refactoring candidates. The difficulty lies in disambiguating, ranking, and filtering these candidates in such a way that only the valid candidates remain. This is done using a similarity metric to calculate the similarity of the bodies of an old and new entity in a refactoring candidate.

3. SIMILARITY METRICS

For each of the refactoring candidates, we compare the bodies of the old entity and the new entity. If these bodies are not equal then we use a similarity measure to determine whether the bodies are similar in a way that it is likely that the external behavior did not change. The main assumption here is that the more similar the body of the old entity is to the body of the new entity, the higher is the probability that the behavior of both artifacts is the same and thus that the refactoring candidate really describes a refactoring. Originally, the technique used CCFinder, a token-based code clone detection tool. In this paper we try to find out how heavily the results depend on the choice of similarity measure. We therefore plug in two other: JCCD, an AST-based clone detector, and shingles, a text-based technique.

Code clone detection is an established technique for identifying similar pieces of code in one or more files. In the specific scenario of refactoring detection, however, only clones between the bodies of two entities are interesting. If these bodies are not clones of each other, it is still interesting whether parts of the bodies are clones.

3.1 CCFinder

CCFinder is a scaleable token-based tool for detecting code clones in various programming languages [11]. It was used in the original technique by Weißgerber and Diehl as a code clone detection tool to determine the similarity between the body of an old entity and the body of a new entity in a refactoring candidate [22]. They configured CCFinder to find pairs of similar Java code fragments j_1, j_2, such that j_1 is equal to j_2 or if j_1 can be transformed into j_2 by performing one or more of the following operations: adding or deleting white spaces, adding, deleting or changing comments, changing visibility, adding or removing the package name, consistently renaming variables, consistently renaming method names, consistently renaming references to member names, or consistently renaming types.

One problem is that when the two code fragments are not clones, then it could still be that token sequences within the bodies could be clones. To take this into account, for each refactoring candidate one can count how many tokens of the body of the old entity are cloned in the body of the new entity. Fortunately, for two code fragments j_1 and j_2, CCFinder does not only determine if j_2 is a clone of j_1. It also identifies all substrings of j_1 and j_2 that are clones of one another. This is used in the similarity metric `CloneFraction`, which was precisely defined by Weißgerber in [20].

3.2 JCCD

JCCD is an AST-based code clone detection API that is based on a generic pipeline model [2]. This model coordinates the interplay of all required steps in a code clone detection process. By combining and parameterizing predefined API components as well as by adding new components, JCCD facilitates to build custom code clone detectors. As JCCD is highly configurable and extensible, we configured it differently for each refactoring kind. For instance to check if a particular RenameMethod candidate is a real candidate, we can ignore the method declaration and the method name; to check if a HideMethod candidate is a real candidate, we can ignore the modifiers of the method, etc. We instantiated the JCCD pipeline in such a way that Weißgerber's `CloneFraction` metric could also be used.

3.3 Shingles

The third way that we used to measure the similarity of method bodies is to use shingles—a text-based approach. In natural language processing a *w-shingling* is a set of unique *shingles* —subsequences of adjacent tokens in a document— that can be used to gauge the similarity of two documents [3]. The 'w' represents the window used to create the shingles, it denotes the number of tokens for each shingle. The resemblance of two documents A and B can be expressed as the ratio of the magnitudes of their shinglings' intersection and union.

In our technique, the shingles similarity takes two strings of program code (s_1, s_2) and tokenizes these strings in order to build *2-shinglings* (whith $w = 2$ we create shingles of size 2) from these tokens. Let $SH(s_1)$ be the set of unique shingles for s_1 and $SH(s_2)$ for s_2. The shingles similarity between these two strings can then be defined as

$$\texttt{ShinglesSimilarity}(s_1, s_2) = \frac{|SH(s_1) \cap SH(s_2)|}{|SH(s_1) \cup SH(s_2)|}$$

In other words, the shingles similarity is computed by the ratio of the number of common shingles and the number of all shingles.

3.4 Disambiguation and Filtering

The metrics described above are used to rank the refactoring candidates. They are used to (i) identify ambiguous candidates (i.e. several refactoring candidates on the same entity) and (ii) to filter out candidates that are likely to be wrong.

It is possible that several refactoring candidates work on the same entities. There are two kinds of ambiguous refactoring candidates: two refactoring candidates $(s_1, k_1, s_1')_t$ and $(s_2, k_2, s_2')_t$ are *source ambiguous* if and only if $s_1 = s_2$ and $s_1' \neq s_2'$ meaning that both refactoring candidates have the same source, but different targets. Analogously, we can define *target ambiguousness*, where two refactoring candidates have the same target but different sources. The entire set of refactoring candidates for a transaction RC_t can now be split into several sets of refactoring candidates that are (target) ambiguous. In these sets of ambiguous refactoring candidates we need to define a ranking. We do this based on the similarity measures defined above. The more similar the body of the old entity is to the body of the new entity, the higher the refactoring candidate is ranked. Each refactoring candidate in a set of n ambiguous candidates is given a natural number between 1 and n. In this ranking the candidate with the lower natural number is better than one with the same target s, but that is ranked with a larger number.
With this ranking we can filter out the worst candidates. In total we can filter on three metrics:

Disambiguation mode $dis \in \{\texttt{none}, \texttt{b1}, \texttt{b2}\}$**:** The `b1` and `b2` filters are based on the ambiguousness ranking. `b1` only allows the refactoring candidate that is ranked best of the ambiguous refactoring candidates. `b2` only allows the best two to pass the filter.

Similarity threshold $thr \in \{\mathbf{0\%}, \mathbf{1\%}, \cdots, \mathbf{100\%}\}$**:** The second filter is to use a threshold on the similarity metric, i.e., we allow only those candidates that have a similarity measure higher than a given threshold t. In all three metrics, the similarity is a number in the range $[0, 1]$, where 1 indicates that the two bodies $\texttt{body}(s)$ and $\texttt{body}(s')$ are identical.

Signature matching $sig \in \{\texttt{strong}, \texttt{weak}\}$**:** The final filter is to only allow either strong or weak (all) candidates.

Thus, the whole refactoring detection process can be considered as a function $\texttt{rede}(sig, dis, thr)$ which yields a set C of refactoring candidates.

4. EVALUATION

In this section we evaluate the similarity metrics described in Section 3 in order to find out what their effect is on the signature-based refactoring detection technique. We do this with a replication experiment, where we replicate the results obtained by Weißgerber and Diehl. However, where they only used CCFinder to calculate the similarity of two entities in a refactoring candidate, we extend this experiment by plugging in the two other similarity metrics. Thus, the extended refactoring detection method $\texttt{rede}(sim, sig, dis, thr)$ gets a fourth argument $sim \in \{\texttt{CCFinder}, \texttt{shingles}, \texttt{JCDD}\}$.

During the entire experiment we kept two basic questions in mind: (i) How big is the influence of any similarity metric within the signature-based refactoring detection? (ii) How do the similarity metrics differ?

4.1 Experimental Setup

For the replication experiment we selected four software projects out of seven which were already analyzed in the study by Weißgerber [20]. In order to get a representative case selection we considered two main aspects: Firstly, we want to analyze projects of different levels of maturity. This is measured by the amount of transactions. Secondly, we want to analyze projects that showed varying performances in the previous study. The software archives we selected on this basis is shown in Table 2.

Table 2: Basic data of the evaluated projects.

Project	# Txns	Time Frame
Azureus	10664	Jul 10 2003 – Feb 12 2007
Jfreechart	2412	Oct 18 2001 – Feb 14 2007
Jftp	209	Jan 25 2002 – Mar 23 2003
Tomcat 3	4158	Oct 09 1999 – Nov 21 2004

A lot of time was invested by Weißgerber and his students to manually gather all documented refactorings of these software archives. In order to reuse this data in our Quality Assessment (see Section 4.8) we considered the exact same time frames. Additional data of the previous study was not required during our replication.

In our study we used an improved version of Weißgerber's refactoring detection library and extended it with JCCD and shingles. We run the analysis separately, for each of these measures and each project. This ensures that no similarity metric could affect the results of another.

We have ignored refactorings concerning fields (RenameField, MoveField, HideField and UnhideField) since a field is an entity with only a signature and no body. Therefore using the similarity measures on the body of a field is useless.

4.1.1 Filter Options

As explained in Section 3.4 the refactoring detection process provides three main filter options. This results in 600 possible combinations of (sig, dis, thr) for one similarity metric. Which leads us to question: What combinations are useful for the comparison of the three similarity measures? Weißgerber only used 7 combinations in his evaluation. Our first idea was then to take only these combinations into account. If we did this, then every similarity measure uses the same threshold value. The problem is however, that a threshold for one particular similarity metric can mean something entirely different for another one. On the other hand, the strong/weak and ambiguity settings have the same effect regardless of the similarity measures. This results in 6 million possible combinations of $(sig, dis, thr_{\texttt{CCFinder}}, thr_{\texttt{shingles}}, thr_{\texttt{JCCD}})$. Therefore, it is necessary to have an objective method to adjust the thresholds $thr_{\texttt{CCFinder}}$, $thr_{\texttt{shingles}}$, and $thr_{\texttt{JCCD}}$ of each similarity metric independently, in order to get comparable candidate sets.

4.1.2 Adjustment of the Similarity Thresholds

The idea is to use a number x of candidates as a calibration value and to give each similarity metric the chance to present its x best candidates. As a result, we obtain comparable candidate sets, that reflect the characteristics of each similarity metric. More precisely, we implemented a function $\mathsf{minthreshold}(sim, sig, dis, x) = thr$ such that $|x - |\texttt{rede}(sim, sig, dis, thr)||$ is minimal. To actually compare the different similarity metrics, we use a range of values for x as described in the next subsection.

4.1.3 Threshold List

For each project we first compute 100 subsequent calibration values $x_1, \cdots, x_{100}$ to be used as predefined numbers of refactoring candidates. Then, for each filter combination (sig, dis) we created a threshold list with 100 calibrated threshold triples of the form $(thr^{x_i}_{\texttt{CCFinder}}, thr^{x_i}_{\texttt{shingles}}, thr^{x_i}_{\texttt{JCCD}})$ where $thr^{x_i}_{sim} = \mathsf{minthreshold}(sim, sig, dis, x_i)$.

Since a threshold of 0% is equivalent to no filter, a threshold of 0% lets all refactoring candidates pass. A threshold of 100% is the most restrictive, but might result in a different amount of remaining candidates for each similarity threshold. Thus, the range of all calibration values that can be reached by adjusting the threshold for each similarity metric is given by the interval $[\max_{sim} |\texttt{rede}(sim, sig, dis, \texttt{100\%})|, \cdots, \min_{sim} |\texttt{rede}(sim, sig, dis, \texttt{0\%})|]$.

In other words, we get the minimal and maximal number of candidates for each similarity metric and chose this as the lower and upper limit that can be reached by each similarity metric. This range is divided into 100 equal parts to get the different calibration values.

4.2 Strong and Weak Candidates

In this section we characterize, how the influence of the similarity metrics varies when using the `weak` or `strong` filters. At first we computed for each entry in the threshold list (see Section 4.1.3) the overlaps of the related candidate sets. For example, the overlap of all three candidate sets is

$$\left| \bigcap_{sim} \texttt{rede}(sim, sig, dis, thr_{sim}^{x_i}) \right|$$

Next we created charts with the calibration values on the x-axis and the related overlaps on the y-axis. Figure 1 shows two such diagrams for JFREECHART. A closer look reveals that in both filters a lower calibration value leads to a lower overlap. This is because for a lower number of candidates, we have higher (more restrictive) thresholds. We can also see that this effect is more accentuated for the `weak` filter.

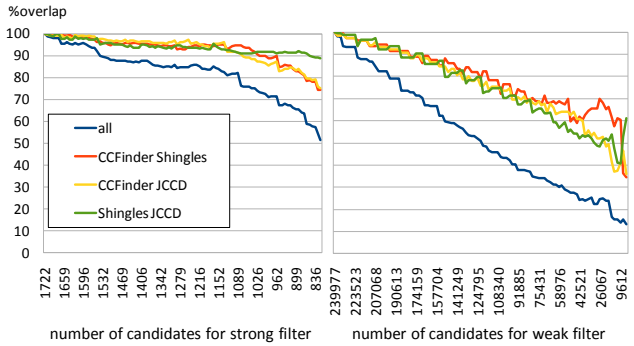

Figure 1: Overlap when using only the `strong` and `weak` filters for JFREECHART.

Table 3: Minimal overlap for weak and strong candidates.

	AZUREUS	JFREECHART	JFTP	TOMCAT 3
weak	11.6%	13.3%	22.8%	2.1%
strong	69.3%	51.5%	74.3%	81.2%

Since the curves for the `strong` and `weak` filters show a similar behavior, we illustrate the difference between these filters using only the minimal overlap values in Table 3. We do not use the maximal overlap value, since this is the same for both filters. We can see that there is a big difference between the overlap of the results obtained with the `strong` filter and the results obtained with the `weak` filter. For instance, in the TOMCAT 3 project the minimal overlap of the `strong` filter is 81.2%, this means that we can choose any calibrated threshold triple and yet their candidate sets have at least an overlap of 81.2%. This suggests that in this case it is not as important which similarity metric is used. In contrast, in the same project the minimal overlap of the `weak` filter is 2.1%. This suggests that for the `weak` filter each similarity metric has a big influence on the candidate set. Therefore, the selection of a particular similarity metric is a significant configuration when using the `weak` filter.

Since we found that for the `strong` filter the similarity metrics have a smaller influence on the candidate set, we will focus on the `weak` filter in the rest of the evaluation.

Table 4: Overlap of all candidates with a similarity threshold of 0% for each metric.

	AZUREUS	JFREECHART	JFTP	TOMCAT 3
b1	60.9%	37.1%	33.3%	49.0%
b2	47.9%	39.4%	30.7%	43.4%

4.3 Ambiguous Candidates

In this section we investigate the relation of similarity metrics and disambiguation. With a threshold of 0% and without disambiguation for both the `weak` and the `strong` filter the overlap is at 100%. In this case, all candidates pass the filter no matter how similar the entities of the candidates are. Thus, at this point, the similarity metrics have no influence.

As we described in Section 3.4, several refactoring candidates are ambiguous. On average, without filtering, each entity refers to about 90 ambiguous refactoring candidates. By using the `b1` filter only the candidates that are ranked best in their sets of ambiguous refactoring candidates will pass the filter. Since the similarity metric is also considered in the ranking of ambiguous candidates, the overall results of the `b1` filter can be very different. With a threshold of 0% but using disambiguation (`b1` or `b2`) the overlap is considerably below 100%, as can be seen in Table 4.

Furthermore, in this table we can see that the biggest overlap can be found in AZUREUS2 with the `b1` filter. This means that all similarity metrics came to a similar ranking of the ambiguous candidates. In JFTP, on the other hand, using the same filter, the similarity metrics have ranked the ambiguous candidates more differently and thus the overlap is much smaller.

Another interesting observation is that for all projects (except for JFREECHART) , the overlap when using the `b2` filter is much smaller than when using the `b1` filter. This can be explained by the fact, that `b2` allows the two top-ranked refactoring candidates while `b1` only allows the top one. As a consequence, the candidate set of the `b1` filter is a subset of the one of the `b2` filter. Nevertheless, the overlap in the `b2` filter is smaller, since in most cases, each similarity metric also uses a different ranking for the second best candidate.

All in all, with the `b1` or `b2` filters, each similarity metric has a significant influence on the filtering process. While we computed all the data and figures for both filters, in the following we will only discuss our findings for the `b1` filter. For the above reasons, they are similar for the `b2` filter.

By focussing on `weak` signature matching and the `b1` disambiguation mode, we can take the two points into account where the similarity metrics have an influence on the refactoring detection process: Firstly, the ranking of the candidates for which we use the `b1` filter; And secondly, the similarity between the two entities in a refactoring candidate for which we use a similarity threshold.

4.4 Correlation of the Similarity Thresholds

We showed in previous sections, that the effect of a similarity metric can be influenced by adjusting the similarity threshold. In this section we investigate if we can find a correlation between adjusting the similarity threshold and the effect of the similarity metric. We measure this correlation by adjusting the similarity threshold and observing

the changes in the number of candidates. We have already learned two facts from previous sections: first, using the same threshold value for each similarity metric leads to a different number of candidates and second, by increasing the threshold values, the number of candidates gets smaller.

When using the CCFinder metric, we observed in all four projects that the number of candidates decreases linearly when increasing the threshold. With shingles and JCCD, on the other hand, we observed threshold ranges in which the number of candidates decreased more rapidly. Thus, the adjustment of the similarity threshold could have a bigger effect on the results of one similarity metric than on another. In other words, there exists no fixed offset between the threshold values. Rather a small adjustment of the threshold could mean for one particular metric a small and for another a big difference.

Table 5: Number of candidates for `b1` by a common threshold value of 100%.

	CCFinder	JCCD	shingles
Azureus	9029	7050	8847
Jfreechart	1105	733	724
Jftp	191	195	208
Tomcat 3	3532	2773	2945

Another fact is that when using a threshold of 100% for each similarity metric, we find a different number of refactoring candidates. This is shown in Table 5, in which we give an overview of the number of candidates produced using the `b1` filter and a threshold of 100%. We can see that CCFinder produces the most candidates whereas JCCD tends to produce smaller result sets. This is a well-known issue for these kinds of clone detectors [15]. In general, most token-based clone detectors like CCFinder have a good recall, yet a weak precision. The opposite is true for tree-based approaches like JCCD. These mostly have a good precision and a weak recall. It seems that this property of CCFinder (to return more code clones than JCCD) was handed over to the refactoring detection. This applies analogously to JCCD and shingles.

4.5 Overlap of the Similarity Metrics

In the previous sections we have mostly been concentrating on our first main research question: How influences each similarity metric the results of the signature-based refactoring detection, and how can this effect be influenced with some adjustments? In the next parts we investigate how the three similarity metrics differ within this signature-based approach, and thus, formulate an answer for our second research question: How do the similarity metrics differ?

In this section, we investigate the overlap of the candidate sets for each calibrated threshold triple using the `b1` filter. To realize this, for each calibration value from the threshold list (see Section 4.1.3) we computed the overlaps for all three similarity metrics and additionally the overlaps for all pairs of metrics. Figure 2 gives an overview of these overlaps.

The first impression is that all charts look quite similar. It is however peculiar that most overlap curves have an S-shape (decrease, increase, decrease) and that the biggest overlap is achieved for small but not for the smallest calibration values (i.e. the highest threshold values). To explain this we have to remember the effect of the `b1` filter. As described in

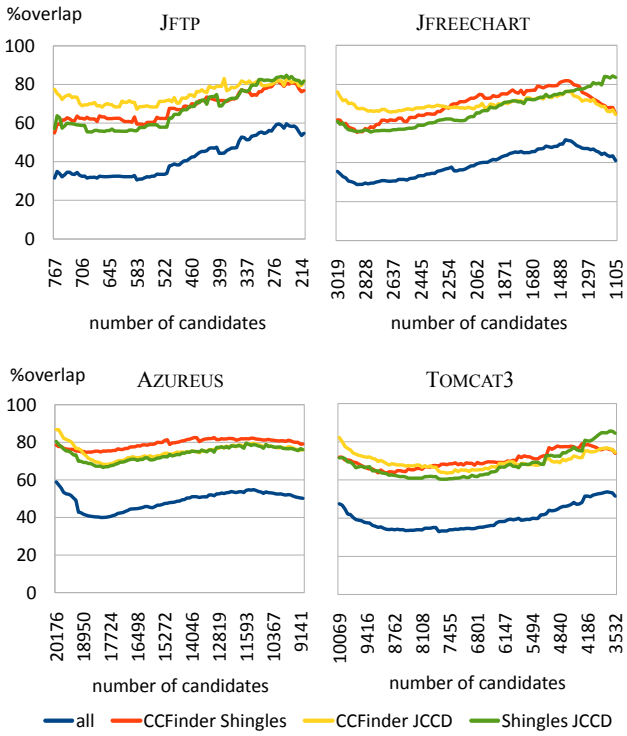

Figure 2: Overlap of all metrics and all thresholds by using the `b1` filter.

Section 4.3 each similarity metric uses another ranking for ambiguous candidates. Note that this ranking is fixed for all threshold values. One thought is that the overlap of all similarity metrics should not become bigger when increasing the threshold values, since a candidate set obtained with a particular threshold is a subset of a candidate set using a smaller threshold. This is however not the case, because at the threshold of 0%, many of the best ranked candidates might still have a small similarity. Thus, by increasing the thresholds, these worse candidates will be pruned and in the end, the different result sets have a bigger overlap. Another fact is that the influence of each similarity metric grows quite slowly, leading to a point in which all metrics have an equally significant influence, and thus, the point with the biggest overlap. After that, each metric influences the results in its own direction.

A closer look at the diagrams also reveals, that the overlaps of two similarity metrics are more balanced in the bigger projects Azureus and Tomcat 3. In the other two projects these overlaps have a bigger variation of about 20%. Nevertheless, if we take the overlap of all metrics into account we can determine that all metrics have a much lower overlap than only two metrics. This means that any two metrics have quite a big subset of candidates which is not present in the candidate set of the third one.

To conclude, we have seen that the selection of a particular metric can significantly influence the direction of the results. However, none of these three similarity metrics was able to change the results in something completely different. There is still a common candidate basis of 30–40% which can be found in the results of each similarity metric.

4.6 Differences of the Similarity Metrics

To get a better understanding why a particular similarity metric filters out a candidate that can still be found in the result set of another metric, we selected for each project the calibrated threshold triple for which the overlap of all candidate sets is the largest. So we expect that the candidates within the complements (the set of candidates that are rejected by the other two metrics) reflect the characteristics of a similarity metric best.

Figure 3 shows Venn diagrams of the candidate sets for each similarity metric considering the selected thresholds for each project. As we can see, in each project the proportions of the sets are quite similar to another. This illustrates again the previous mentioned claim that there exists a big basis of common candidates (more than 50%), but also that none of the other intersections is empty, they contain at least 2.5%.

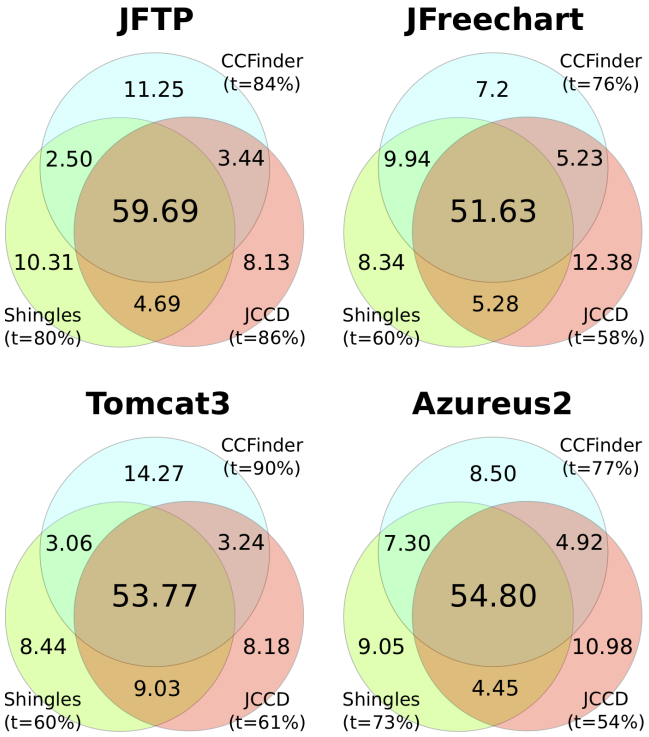

Figure 3: Biggest possible overlap (in %) of all similarity metrics compared to other threshold combinations by using the `b1` filter.

We have manually inspected a total of 328 candidates (about 8–13% of all candidates) from the complements of JFTP (95 candidates) and JFREECHART (233 candidates). In general, we observed many candidates with very small code fragments. In total 195 candidates (59% of the inspected candidates) refer to code fragments which have less than three lines of code. However, we noticed that the computed similarity metrics were rather random. This is because a small difference in a token can lead to a low similarity value and thus such a candidate is unable to meet the threshold value. By ignoring smaller code fragments the common candidate basis will even increase (e.g. in JFREECHART from 51.63% to 62.6%, or in TOMCAT3 from 53.77% to 82.32%).

Furthermore, some candidates did not represent real refactorings. For example, two unrelated getter-methods could be rated as similar, since both clone detectors have a big tolerance for identifiers. Additionally, we found that 83 candidates (25% of all inspected candidates) had only a small difference, and thus, for some thresholds this was enough to be passed and for others not.

Another interesting observation is that JCCD is more stable against clojust nes with missing lines or clones with a permutation of the statements than when we use JCCD in a standalone way (i.e. to detect clones and not within this refactoring detection framework). This is because JCCD returns a lot of small clones which are merged by the computation of the `CloneFraction`.

We also found candidates that looked very similar and were only found by shingles. The only difference was that in one code fragment several lines were commented out. While JCCD and CCFinder ignore comments, the shingles metric takes them fully into account. Another characteristic of shingles is that the smaller code fragments tend to be ranked more dissimilar and bigger code fragments more similar. The reason for this is that the probability of new unique shinglings in large documents is much lower than in smaller documents, because the amount of different tokens and their permutation possibilities are limited. Thus, we found two code fragments which were rated as 100% similar by shingles, but in the second code fragment a whole part was copied to another position in the code.

All in all, subtle peculiarities of the similarity metrics can make themselves felt in the end results.

4.7 Correlation of the Recall Values

In this section, we investigate how the recall values of each similarity metric relate to the calibration values. With that we want to find out if one particular similarity metric leads to a higher recall value than another one for a given number of candidates.

We have no information about the exact refactorings that have been performed during the development of the four projects and thus we have no way of calculating the exact recall of the signature-based refactoring detection. We can, however, estimate the recall based on the documented refactorings (DR). Weißgerber and his students scanned the commit messages in the source repositories for log messages mentioning refactorings. They then manually inspected the exact source code changes that were made during this refactoring to categorize them correctly. The documented recall can then be calculated as $drecall = \frac{|RC \cap DR|}{|DR|}$ where RC is the set of computed refactoring candidates.

For all four projects, Table 6 lists all documented refactorings broken down into three refactoring categories. We can see, that most documented refactorings are method refactorings followed by class refactorings and interface refactorings. Thus, the overall documented recall value includes a specific weighting based on the distribution of the documented refactorings. Within this exploration we have excluded JFTP because for this case we only have a total of 6 documented refactorings and this will therefore not produce any meaningful results.

By analyzing the drecall values for each calibrated threshold triple we find that the drecall is similar for each threshold of the triple. Which is a surprising fact, since we mentioned in a previous section, that JCCD has a high precision and CCFinder has instead a high recall. It seems that this is not reflected in the drecall of the signature-based refactoring de-

Table 6: Number of documented refactorings

Project	ClassRefs	InterfaceRefs	MethodRefs
Azureus	249	59	243
Jfreechart	28	2	266
Jftp	1	0	5
Tomcat 3	75	20	271

tection. Only in Azureus we note that the drecall values of JCCD behave a bit different. Just by increasing the threshold value of JCCD from 0% to 1% the drecall of JCCD drops from 92% to 77% while, using the other metrics, the drecall values remain up to 92–95% with higher threshold values. If we compare the numbers of documented refactorings we can determine that Azureus has significantly more class refactorings. It could be that class refactorings can be more reliably detected by using shingles or CCFinder. Nevertheless, this distribution remains the same for the rest of the threshold combinations.

Additionally, we determine that the drecall of the intersection is quite high, and thus, most detected documented refactorings can be found using any of the three metrics. Hence, with a combination of all similarity metrics it could be possible to increase the precision, since the candidate set can be reduced while keeping the drecall more or less equal. The drecall of the union is as expected slightly higher than the drecalls of each similarity metric. Thus, by taking the union candidate set it is possible to increase the drecall and the recall respectively.

4.8 Quality Assessment for each Metric

In this section, we estimate the quality of the results obtained using each of the similarity metrics. We base this estimation on the drecall values and the calibration values. For the estimation we assume that if the drecall value for a particular candidate set remains the same while reducing the number of candidates, then the precision value should increase. In other words, with a fixed number of candidates, the similarity metric that leads to a result with a higher drecall than the other metrics, will also lead to a higher precision. Nevertheless, we are unable to estimate concrete values for the precision.

We have again taken the same result sets as in Section 4.6 and listed their recall values in Table 7. As we can see, most recall values are comparable to each other per project and per refactoring kind. Overall, with these similar recall values, the similarity metrics could lead to comparably high precision values. Thus, we believe that the quality of the results is comparable, independent of the used similarity metric.

We note that the drecall of JCCD in Azureus for the set of class and interface refactorings is much lower than for the other similarity metrics. In detail, JCCD only found 24% of the RenameClass refactorings while shingles has a drecall of 76% and CCFinder 85%. This could support our claim from the previous section that JCCD is weaker for detecting class refactorings. However, this effect can only be observed in Azureus.

On average for method refactorings CCFinder seems to generate the lowest drecall values. As already mentioned by Weißgerber we observed several problems to detect AddParameter and RemoveParameter refactorings. But this does not seem to be a problem of the signature-based approach in general but of the used similarity metric. By selecting a more appropriate similarity metric this can be influenced in a positive way. For example, in the Tomcat 3 case CCFinder has only found 41% of the documented AddParameter refactorings while shingles has found 89% and JCCD 81%.

Table 7: Documented Recall.

RefKind	Project	CCFinder	JCCD	shingles
Class	Azureus	0.94	0.72	0.91
	Jfreechart	0.96	0.96	0.96
	Jftp	1.00	1.00	1.00
	Tomcat 3	0.88	0.89	0.93
Interface	Azureus	0.97	0.76	0.98
	Jfreechart	1.00	1.00	1.00
	Jftp	—	—	—
	Tomcat 3	1.00	1.00	1.00
Method	Azureus	0.68	0.75	0.79
	Jfreechart	0.70	0.71	0.69
	Jftp	0.60	0.60	0.60
	Tomcat 3	0.75	0.82	0.85

Surprisingly, on average the simplest similarity metric, shingles, leads to the best drecall values for this particular threshold combination.

All in all, this is only a rough quality assessment which is strongly based on the documented refactorings. In many cases all similarity metrics seem to have a comparable quality. Additionally, as we have seen before, the results are largely overlapping, and thus we do not expect a large difference in the quality. It is however not entirely clear why particular peculiarities of the similarity metrics in some cases shine through and in other cases they do not.

4.9 Computation Time

We ran the experiments on a server with four processors (eight cores, 2Ghz) and 32GB of memory. The computation times for each of the analyses (on the four cases), using the three metrics is shown in Table 8. This table shows the average computation time per transaction. Note that these runtimes include the time needed for the preprocessing step of the signature-based analysis. However, the computation time for the preprocessing step is equal for each of the similarity metrics. For instance in the case of Jfreechart the preprocessing step took about two hours (or three seconds per transaction). So in reality the runtimes of calculating the similarity metrics are lower than shown in this table, the difference between the runtimes of the three different metrics, however, remains the same.

We can observe that CCFinder is much slower than the other two similarity metrics and the shingles similarity metric (the simplest) is calculated fastest. As in the original implementation, CCFinder needs to be run through `wine`[2] which adds about five milliseconds per refactoring candidate to the total runtime, yet even when we take this into

[2]Wine is a program that allows us to run Windows applications on Linux or Mac OSX.

Table 8: Average Runtime per Transaction

Project	# Txns	CCFinder	JCCD	shingles
Azureus	10642	20.1s	10.8s	6.1s
Jfreechart	2411	29.2s	7.0s	4.0s
Jftp	208	55.3s	12.9s	5.5s
Tomcat 3	4148	45.0s	8.1s	4.1s

account CCFinder is still outperformed by the other two. This confirms the conclusions drawn in [2], where JCCD was compared to CCFinder and JCCD was found to be faster.

5. THREATS TO VALIDITY

In this section, we identify factors that may jeopardize the validity of our results. Consistent with the guidelines for case studies research (see [16, 24]) we organize them into four categories.

Construct validity. (*Do we measure what was intended?*) We compared three different similarity measures. We can still extend this experiment with other similarity measures. A recent survey by Roy and Cordy showed more than forty different approaches to code clone detection. These can be roughly categorized by the kind of information they process: strings, tokens, abstract syntax trees, program dependence graphs, metrics, or hybrid approaches [15].

Internal validity. (*Are there unknown factors which might affect the outcome of the experiment?*) For each of the four cases, we used a set of documented refactorings to estimate the recall. These documented refactorings are based on the commit messages in the log of the source repository. It is very likely that this is only a subset of all the performed refactorings, since sometimes a developer might find a refactoring not relevant to mention in the commit message, or the refactoring might be so complex that the developer did not realize he was refactoring the code. The real recall values might still vary from our estimated recall based on the documented refactorings. The estimated recall however is sufficient to compare the different similarity metrics.

External validity. (*To what extent is it possible to generalize the findings?*) We performed our experiment on four open source Java projects of varying sizes and application domains. We therefore cannot claim that these findings can be held true for other programming languages or for projects of a larger scale. Nevertheless, while the optimal configurations for the clone detection tools differ across projects, there is a clear trend that a lot of the refactorings are found by all three metrics with a comparable drecall value.

Reliability. (*Is the result dependent on the researchers and tools?*) We relied on tools of our own making. The tool that we used to detect the refactorings is the same one as used by Weißgerber and Diehl [22] and we have extended that with a similarity metric calculated by JCCD and with shingles.

We used the documented refactorings found and analyzed by Weißgerber and his students, and since they are no experts in the development of the open source cases, there could be some misclassifications in the refactorings.

6. RELATED WORK

There has already been quite some work on techniques to find refactorings. Many of which use code clone detection as a basis for their algorithm:

Danny Dig et al. have developed an open source refactoring detection tool RefactoringCrawler, which uses a combination of syntactical and semantical analysis to identify refactorings [5, 6]. Their approach is very similar to the one by Weißgerber and Diehl [22] (which we explained in Section 2). The difference lies in the fact that Dig first uses the (shingles) similarity metric to identify possible refactoring candidates and then uses the signature based analysis to filter out the bad candidates.

Van Rysselberghe and Demeyer use clone detection on two versions to look for a decrease in the number of clones. Since many refactorings are aimed at eliminating software clones a decreased number of clones in one version would suggest that refactorings were performed [19].

Other approaches don't use clone detection, but do show promising results:

Xing and Stroulia have developed an algorithm that compares two subsequent versions of a system at the design level. Their UMLDiff algorithm is capable of detecting some basic structural changes in the system such as the addition, removal, renaming or moving of UML entities. More complex structural changes can be found using a suit of queries that try to find a composition of elementary changes [23, 17].

Demeyer et al. have developed a set of heuristics to identify refactorings. Each heuristic is a combination of change metrics to reveal refactorings of a certain kind. What they do is compare different source code metrics on subsequent versions of a system. However code metrics do not provide sufficient information to pinpoint which elements were involved in the refactoring [4].

Prete et al. proposed a refactoring detection technique, which is stronger than all previous techniques because they not only detect primitive refactorings (which all previous techniques can do to some extent) 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) [13].

7. CONCLUSIONS

At MSR 2010 the paper by Robles [14] and the invited talk by Juristo and Vegas [10] on non-identical replication led to a lot of discussion on the lack of replicated experiments in the area of mining software repositories. In this paper we presented a replication experiment, where we took a refactoring detection technique that used a specific code clone detector (CCFinder) and investigated what the effect was of plugging in two other similarity measures.

We found that the similarity metric has an influence on the signature-based refactoring detection in several places. When using the `strong` filter, the similarity metrics have a much smaller influence, than when using the `weak` filter. This is because the `strong` filter is already very selective. The similarity metrics also have an influence in the disambiguation of the refactoring candidates, since the ambiguous candidates are ranked using the similarity metrics and then the `b1` or `b2` filter is used to only allow the highest ranked candidates. However depending on the similarity metric, the ranking of the refactoring candidates differs. Even though

each of the similarity metrics have an influence on the result sets, there is still a common basis of refactoring candidates that can be found in the results of each similarity metric. The little differences in the result sets that do occur, are due to subtle peculiarities of the similarity metrics, like the fact that shingles takes comments into account, or the fact that many of the refactoring candidates involve entities with very short bodies (e.g., getter and setter methods). We also looked into the quality of the result sets and found that in many cases the similarity metrics show a comparable quality. When we look at the computation times needed to calculate the three similarity metric, we see that shingles outperforms the other two and that CCFinder is by far the slowest.

Acknowledgments

Peter Weißgerber helped us to understand his signature-based refactoring detection library. He also provided all relevant data to perform the replication experiment. Fabian Beck gave helpful comments on a draft of this paper.

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] B. Biegel and S. Diehl. Highly configurable and extensible code clone detection. In *WCRE'10: Proc. Working Conference on Reverse Engineering*, 2010.

[3] A. Broder. On the resemblance and containment of documents. In *SEQUENCES'97: Proc. Compression and Complexity of Sequences*. IEEE Computer Society, 1997.

[4] S. Demeyer, S. Ducasse, and O. Nierstrasz. Finding refactorings via change metrics. In *OOPSLA'00: Proc. Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2000.

[5] D. Dig, C. Comertoglu, D. Marinov, and R. Johnson. Automated detection of refactorings for libraries and frameworks. In *WOOR'05: Proc. Workshop of OO Reengineering*, 2005.

[6] D. Dig, C. Comertoglu, D. Marinov, and R. Johnson. Automated detection of refactorings in evolving components. In *ECOOP'06: Proc. European Conference on OO Programming*, 2006.

[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. Error detection by refactoring reconstruction. In *MSR'05: Proc. International Workshop on Mining Software Repositories*, 2005.

[9] J. Henkel and A. Diwan. Catchup!: capturing and replaying refactorings to support api evolution. In *ICSE'05: Proc. International Conference on Software Engineering*, 2005.

[10] N. Juristo and S. Vegas. Using differences among replications of software engineering eperiments to gain knowledge. In *ESEM'09: Proc. International Symposium on Empirical Software Engineering and Measurement*, 2009.

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

[12] W. F. Opdyke and R. E. Johnson. Refactoring: An aid in designing application frameworks and evolving object-oriented systems. In *SOOPPA '90: Proc. Symposium on Object-Oriented Programming Emphasizing Practical Applications*, September 1990.

[13] K. Prete, N. Rachatasumrit, N. Sudan, and M. Kim. Template-based reconstruction of complex refactorings. In *ICSM'10: Proc. International Conference on Software Maintenance*, 2010.

[14] G. Robles. Replicating msr: A study of the potential replicability of papers published in the mining software repositories proceedings. In *MSR'10 Proc. Working Conference on Mining Software Repositories*, 2010.

[15] C. K. Roy, J. R. Cordy, and R. Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming*, 74:470–495, May 2009.

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

[17] C. Schofield, B. Tansey, Z. Xing, and E. Stroulia. Digging the development dust for refactorings. In *ICPC '06: Proc. International Conference on Program Comprehension*, 2006.

[18] Q. D. Soetens and S. Demeyer. Studying the effect of refactorings: a complexity metrics perspective. In *QUATIC'10 Proc. International Conference on Quality in Information and Communications Technology*. IEEE Computer Society Press, 2010.

[19] F. Van Rysselberghe and S. Demeyer. Reconstruction of successful software evolution using clone detection. In M. W. G. Tommi Mikkonen and M. Saeki, editors, *IWPSE'03: Proc International Workshop on Principles of Software Evolution*, pages 126–130. IEEE Computer Society, September 2003.

[20] P. Weißgerber. *Automatic Refactoring Detection in Version Archives*. PhD thesis, University of Trier, 2009.

[21] P. Weißgerber and S. Diehl. Are Refactorings less error-prone than other changes? In *MSR'06: Proc. International Workshop on Mining Software Repositories*, 2006.

[22] P. Weißgerber and S. Diehl. Identifying refactorings from source-code changes. In *ASE '06: Proc. International Conference on Automated Software Engineering*, pages 231–240, Washington, DC, USA, 2006. IEEE Computer Society.

[23] Z. Xing and E. Stroulia. Refactoring detection based on umldiff change-facts queries. In *WCRE '06: Proc. Working Conference on Reverse Engineering*, pages 263–274, Washington, DC, USA, 2006. IEEE Computer Society.

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

[25] T. Zimmermann and P. Weißgerber. Preprocessing CVS data for fine-grained analysis. In *MSR'04: Proc. International Workshop on Mining Software Repositories*, 2004.