An Exploratory Study of Identifier Renamings

Laleh M. Eshkevari
École Polytechnique de Montréal, Canada
laleh.mousavi-eshkevari@polymtl.ca

Venera Arnaoudova
École Polytechnique de Montréal, Canada
venera.arnaoudova@polymtl.ca

Massimiliano Di Penta
University of Sannio, Italy
dipenta@unisannio.it

Rocco Oliveto
University of Molise, Italy
rocco.oliveto@unimol.it

Yann-Gaël Guéhéneuc
École Polytechnique de Montréal, Canada
yann-gael.gueheneuc@polymtl.ca

Giuliano Antoniol
École Polytechnique de Montréal, Canada
antoniol@ieee.org

ABSTRACT

Identifiers play an important role in source code understandability, maintainability, and fault-proneness. This paper reports a study of identifier renamings in software systems, studying how terms (identifier atomic components) change in source code identifiers. Specifically, the paper (i) proposes a term renaming taxonomy, (ii) presents an approximate lightweight code analysis approach to detect and classify term renamings automatically into the taxonomy dimensions, and (iii) reports an exploratory study of term renamings in two open-source systems, Eclipse-JDT and Tomcat. We thus report evidence that not only synonyms are involved in renamings but also (in a small fraction) more unexpected changes occur: surprisingly, we detected hypernym (a more abstract term, *e.g.*, size vs. length) and hyponym (a more concrete term, *e.g.*, restriction vs. rule) renamings, and antonym renamings (a term replaced with one having the opposite meaning, *e.g.*, closing vs. opening). Despite being only a fraction of all renamings, synonym, hyponym, hypernym, and antonym renamings may hint at some program understanding issues and, thus, could be used in a renaming-recommendation system to improve code quality.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics—*Product Metrics*

General Terms

Languages, Human Factors, Experimentation

Keywords

Identifier renaming, software evolution, mining repositories.

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

1. INTRODUCTION

Identifiers (along with comments) are often the sole reliable source of documentation to support everyday developers' activities. Other forms of documentation are scarce (*e.g.*, design documents) or outdated/unappropriate (*e.g.*, user manuals). In the following, for the sake of simplicity, we refer to the names of any programming entities, *i.e.*, variables, classes/interfaces, attributes, methods, constructors, and formal parameters, as identifiers.

Identifiers reflect the developers' domain model, experience, culture, and personal taste. They capture the developers' understanding as well as solution space model; they are intended to convey key information to the reader supporting the program understanding activity. Identifiers are often composed of elementary terms (English words, abbreviations, jargon terms, acronyms) as in *getPacketCRC*, and should be consistent and concise [5, 13].

As the source code evolves, identifiers evolve too [1] and previous works, such as [13, 19], investigated the evolution of the structure of identifiers, identifier renamings, and the presence and stability of domain terms [11]. In particular, identifiers can be renamed while preserving the compilability of the system ("rename" refactoring) for various reasons, including improving consistency, conciseness, code readability, and making identifiers follow more closely naming conventions. A modification may consist in adding/removing terms, *e.g.*, the term *role*[1] is added to the identifier *list*, thus the identifier becomes *roleList*. Also, a term can be changed into a synonym, *e.g.*, *run* becomes *execute*. In some cases an identifier, or one of the identifier terms, may be changed into one of its hyponym (*i.e.*, a more specific term, *e.g.*, *position* becomes *line*), hypernym (*i.e.*, a more generic term, *e.g.*, *statement* becomes *declarations*), or even antonym (contrary, *e.g.*, *down* becomes *up*).

All the above modifications point to some kinds of domain or program understanding issues. Although the real reasons for renaming identifiers are privy to developers' intent and fall outside the scope of this paper; we study identifier renamings to identify the types of renamings, their frequencies and locations. We concur with [4, 5] that consistent and concise identifiers help in program understanding. Indeed, lacking consistent and concise identifiers may ultimately lead to program comprehension problems and increase fault prone-

[1]All examples are taken from Eclipse-JDT and Tomcat.

ness. Synonyms and homonyms may highlight identifiers lacking consistency and conciseness [13]. Renamings involving synonyms, hypernyms, hyponyms, and antonyms may highlight quality issues [13], possible understanding problems, and point to change- and defect-prone entities.

This paper presents a preliminary investigation of identifier renamings, aiming at characterizing and quantifying renamings in two real world systems, Eclipse and Tomcat. We propose a first taxonomy of renamings, along four orthogonal dimensions: (1) kind of entity whose names changed, *e.g.*, a method or a local variable; (2) kind of semantic change, *e.g.*, antonymy; (3) string distance, *i.e.*, textual similarity between the terms (before and after renaming); and, (4) grammatical change, *e.g.*, noun to verb.

The taxonomy is implemented into an approximate lightweight, scalable, automated approach to identify and categorize renamings. To identify possible renamings, for each CVS/SVN change set, we analyze the system before and after the change, by comparing subsequent file revisions using *diff* and by mapping variable declarations (extracted from a parse tree) within source code files. This analysis phase is linear in the size of the system and outputs pairs of possibly renamed identifiers. To increase the accuracy and minimize false positives, we further filter out the identified renamings where the string distance (Levensthein distance) between the two variables is high and their type is different. Identifier pairs are then split into their respective terms, which are in turn mapped with one another (before and after renaming). Renamed term pairs are finally automatically classified according to the proposed taxonomy using an approach relying on the WordNet [21] upper ontology.

The paper also reports an empirical study investigating renamings taking place over the lives of two widely-used open-source systems: Eclipse-JDT and Tomcat. We collected about 50,000 file revisions for each systems, for a time interval of six years and eight years respectively. After detecting renamings, we performed a thorough manual validation of all the renamings in Tomcat and of a subset of those in Eclipse-JDT, due to the prohibitively-large number of renamings (4,500) in Eclipse-JDT. We report the frequencies and examples of renamings for the various dimensions of our taxonomy. We encountered not only renamings into synonyms but also into hypernyms, hyponyms, and antonyms, justifying the conjecture that renamings may indeed reflect changes to the developers' domain model.

The remainder of the paper is organized as follows. Section 2 presents our taxonomy while Section 3 details our renaming identification approach. Section 4 presents the two case studies and Section 5 reports their results while Section 6 discusses the threats to their validity. Section 7 describes related work. Finally, Section 8 concludes and outlines directions for future work.

2. TERM RENAMING TAXONOMY

This section describes our proposed taxonomy for term renamings. Identifiers are composed of terms and thus they are renamed by adding/removing terms or changing a term into another term. Thus, we can see an identifier renaming as a transformation Tr, which itself is a composition of transformation functions tr_i, where each tr_i operates on terms and maps a term in the original identifier (id_1) to a term in the renamed identifier (id_2). After having identified and classified term renamings as described in the following, we would need a mapping from the term-renaming space to the identifier-renaming space. The identifier-renaming space may have only one dimension, *i.e.*, a label merging all its composing terms renamings or many different dimensions. In this paper, we study term renamings and we leave the combination of term renamings as part of our future work.

We define the taxonomy of term renamings along four orthogonal dimensions, where each dimension characterizes a renaming from a different aspect: D1 from the programming language (in this case object-oriented); D2 from sense relations, as defined in linguistics; D3 from string distance, as adopted by researchers in the field of identifier evolution; and, D4 from grammatical aspects. Each dimension has a finite set of values described in the following subsections.

2.1 D1: Entity Kind

Renaming an identifier necessarily happens or propagates to the identifier declaration so that the system still compiles. In the case of object-oriented programming, the entity whose identifier is renamed can be the name of a class, an interface, a field, a method, a constructor, a formal parameter, or a local variable. All of the above represent the values of dimension D1. We distinguish the entities being renamed because their renamings have different impact on the source code and, consequently, on program comprehension: while renaming a local variable may help ease the developers' understanding of one method, renaming a class potentially means a change to the domain model of the program.

2.2 D2: Semantic

When identifiers are renamed, a few situations may occur. A term may be added, removed, or changed. When a term is added or removed, the meaning that this term carries is added or removed to/from the identifier. Thus, the first two values of this dimension are the following:

- Add a meaning;
- Remove a meaning.

When a term is changed, it can either preserve or change the meaning of the identifier. The following five values consider a renaming when a term is changed.

Keeping the meaning:

- The original and renamed terms have the same meaning. An example is synonymous terms, *e.g.*, *createDialogFields* → *createDialogArea*. Another example is terms that are abbreviations/expansions of each other, *e.g.*, *invocationType* → *invocType*. Similarly, fixing a typographical error will take this value, *e.g.*, *getSimilararity* → *getSimilarity*.
- The original and renamed terms hold a generalization/specialization relation (hyponym/hypernym), *e.g.*, *thrownExceptionSize* → *thrownExceptionLength*, and *getAccessRestriction* → *getAccessRuleSet*.

Changing the meaning:

- The renamed term has the opposite meaning of the original term (antonym), *e.g.*, *disableLookups* → *enableLookups*.

- The renamed and the original terms have a whole part relation (holonym/ meronym); respective examples are *Point* → *Line* and *Line* → *Point*.

- The original and renamed terms have unrelated meanings, *e.g.*, *problemsCount* → *problemLine*.

2.3 D3: String Distance

This dimension assesses the distance between the original and renamed terms. One such measure is the Levenshtein edit distance [16], which we use to compute the number of editing operations to obtain the renamed from the original identifier. For example, the Levenshtein edit distance between *house* and *home* is 3, because *home* is obtained from *house* by removing two characters *us* and adding one *m*. To have comparable Levenshtein distances, we use the normalized edit distance (*nld*) given by:

$$nld(t_1, t_2) = levenshtein(t_1, t_2)/sum(length(t_1), length(t_2))$$

where *levenshtein* computes the Levenshtein distance.

In our empirical study, we consider the distance to be low if $nld(t_1, t_2) \leq 0.40$, high otherwise. The threshold was defined by trial and error. Corresponding examples for each value are *getUnqualifiedTypeName* → *getQualifiedTypeName*, and *fTypeSignature* → *fTypeName*. Also, we count substitution as an edit operation with cost one (and not as a deletion followed by an insertion with cost two).

2.4 D4: Grammatical

When renaming an identifier, the grammatical type of the changed term may change, *e.g.*, from a noun to a verb (as in *assignmentImplicitConversion* → *preAssignImplicitConversion*), from a noun to an adjective (as in *getSelection* → *getSelectedObject*), or yet again from a verb to a noun (as in *preparedAuthenticate* → *preparedCredentials*). Grammatical type changes include all combinations of renamings from/to noun, verb, adverb, adjective. In the case of a term that does not belong to a dictionary, the value will be none, *e.g.*, the renaming *con* → *connect* has value none to verb.

2.5 Example of Use

We illustrate the use of our taxonomy with an example from Eclipse. Consider a renaming of a method *getFieldsReferencedIn* → *getFieldReferencesIn*. The identifiers are split into terms: (*get, Fields, Referenced, In*) and (*get, Field, References, In*). We can observe that there are two term renamings. The first renaming, *Fields* → *Field*, takes the following values on the different dimensions: D1: method, D2: same meaning, D3: low distance ($nld = 0.09$), D4: noun → noun. The values of the second renaming, *Referenced* → *References*, are: D1: method, D2: same meaning, D3: low distance ($nld = 0.05$), D4: adjective → noun.

3. PROPOSED APPROACHES FOR RENAMING IDENTIFICATION AND CLASSIFICATION

Our approach is composed of several steps. For each change committed in a CVS/SVN, it considers the system source code files before and after the change. It then compares files using the *diff* tool, the output of which it merges with file-level parse information to locate modified declarations and identify possible renaming pairs. It then filters renaming pairs to favor precision and reduce the amount of required manual validation. Finally, after Camel Case split, it classifies and reports term renamings.

3.1 Identification

To perform a study of identifier renamings, we download all revisions (belonging to a given time interval of interest) of each file from the versioning systems of a given system. Then, for each file f_i, we analyze each pair of subsequent revisions $f_{i,j}$ and $f_{i,j+1}$ using the approach described below.

Our approach does not perform origin analysis [10]. We focus in the following on renamings that occur inside the same programming entity and same file. Thus, the only class renamings (D1: class) that we detect correspond to inner classes and (non-public) classes declared in a same file. Our future work includes merging our approach with a class evolution mapping approach, *e.g.*, [12], to consider class/file renamings also. In our experience, working at file-commit level mitigates the class renaming effect as each change is analyzed at a finer temporal granularity with respect to analyzing releases. For Eclipse, in [12], it was found that, at release level, class renaming impacts on average less than 5% of classes if the first Eclipse release was excluded.

3.1.1 Mapping Lines of Code

The first step of our approach maps lines of $f_{i,j}$ into lines of $f_{i,j+1}$ using the output of a line differencing tool. Our approach can use any line-based differencing tool, *e.g.*, the Unix *diff* or *ldiff* [3]. In the next study, we use *diff* as we found that it suffices to track the information of interest. The approach works as follows:

- if there is a change (l_x, l_y c l_z, l_k), all lines in the ranges $x - y$ are mapped into lines $z - k$. If the multiplicity is one-to-many, many-to-one, or many-to-many, the mapping is imprecise. For example, `1c1,2` means that line 1 is mapped into lines 1 and 2 in the new revision; then, we keep into account that the subsequent lines will be shifted (up or down) of $z - k - y + x$ positions;

- if a set of lines is deleted (l_x, l_y d l_z), lines $x - y$ of $f_{i,j}$ are mapped to "-" in $f_{i,j+1}$, and the subsequent lines are shifted up of $y - x + 1$ positions;

- if a set of lines is added (l_x d l_y, l_z), lines $y - z$ of $f_{i,j+1}$ are mapped to "-" in $f_{i,j}$, and the subsequent lines are shifted down of $z - y + 1$ positions.

3.1.2 Identifying Declarations

Once we have mapped lines in $f_{i,j}$ and $f_{i,j+1}$, we identify declarations contained in both source code files. We consider (for the Java language) class (CD), interface (ID), method (MD), formal parameter (PD), field (FD), constructor (COD), and local variable declarations (LVD). We build a forest of parse trees from the modified files using the robust, error-tolerant parser of the Eclipse platform. For each declaration, we identify (i) the line where it occurs, its kind (CD, ID...), its name (identifier), and type (if any).

Then, we produce a set of *likely mappings* between declarations by combining the output of this step with the output of the previous step (line mapping). Given two lines mapped between $f_{i,j}$ and $f_{i,j+1}$, *e.g.*, lines 18-21 → lines 20-24, we map any declaration occurring in the line interval 18-21 of $f_{i,j}$ into any possible declaration of the same type occurring in the interval 20-24 of $f_{i,j+1}$.

3.1.3 Filtering the Results

The mappings produced in the previous step surely contain a high number of false positives, *i.e.*, pairs of identifiers that are not renamings of one another. To reduce this number, we apply two heuristics to filter the likely mappings: the *type* heuristic and the *similarity* heuristic.

The *type* heuristic matches FDs, LVDs, or PDs if their types are the same. The advantage of this heuristic is that it works well even if the renaming is complete, *e.g.*, *foo* is renamed into *bar*; its disadvantages are that (i) the type of a variable may change together with its name when a renaming occurs at the same time as other refactorings and (ii) the same line may declare more than one variables of the same type, *e.g.*, *String foo, bar;*.

The *similarity* heuristic can be applied to any declaration to keep only identifier pairs in which, despite the renaming, the identifiers are still textually similar. The similarity is computed using *nld*. This heuristic has the advantages of (i) working even when the type changes and (ii) resolving mappings when there are multiple declarations on the same line. For example, if the line *String win, homePtr;* of $f_{i,j}$ is mapped into line *String window, homePointer* of $f_{i,j+1}$, then the inferred renamings will be *win* → *window* and *homePtr* → *homePointer*. The disadvantage of this heuristic is that it fails to capture the renaming of an identifier into a completely different, unrelated one, *e.g.*, *foo* into *bar*.

More effective filtering strategies could be devised, such as the heuristic suggested in [19], which uses data dependencies. However, as we aim at also tracking methods renamings, such a filtering heuristic would require to build call graphs to inspect and compare call sites of all possibly-renamed methods. Such call graphs would, in turn, require polymorphism resolution. Accurate and precise call graph construction for programs of hundreds of thousands of LOCs is expensive, especially if we must build it for every CVS/SVN committed transaction. Thus, we leave for future work the exploration of the advantages/disadvantages of such sophisticated heuristics.

3.2 Classification

The classification starts with two identifiers, the original (id_1) and the renamed identifiers (id_2), and classifies the renaming of id_1 into id_2 as follows.

3.2.1 Identifier Splitting

First, we split both identifiers into their composing terms and each term is converted into lower cases. For the sake of simplicity, we use a simple Camel Case splitter, which splits identifiers considering the Camel Case separator heuristic, as well as commonly used separators such as underscore "_". The output of this step is, for each identifier, a list of terms, *i.e.*, $t_{1,1}, t_{1,2}, \ldots, t_{1,n1}$ and $t_{2,1}, t_{2,2}, \ldots, t_{2,n2}$. For example, the identifier *getBookingInfo* is split into *get, booking, info*. We could use a more sophisticated identifier separator, such as *Samurai* [8] or *TIDIER* [17]. However, we need in our context a fast splitter, given the high number of identifiers to be treated. Also, in our study, we consider only Java systems, for which a Camel Case splitter exhibits performances similar to other approaches [17]. Hence, we leave to future work the use of more sophisticated splitters and their applications to study renamings in other programming languages than Java, without loss of generality, except for D1.

```
foreach matchType in (exact, string_distance, semantic)
do
    for x ← 1 to n1 do
        if not mapped1[x] then
            y1 ← x, y2 ← x ;
            while y1 > 0 or y2 ≤ n2 do
                foreach y in (y1, y2) do
                    if matching(t1,x,t2,y, matchType) and
                    not mapped2[y] and y > 0 and y <= n2
                    then
                        mapped1[x] ← y ;
                        mapped2[y] ← x ;
                    end
                end
                y1 − −, y2 + + ;
            end
        end
    end
end
```

Algorithm 1: Term mapping algorithm mapping the $n1$ terms of id_1 into the $n2$ terms composing $id2$.

3.2.2 Mapping Identifier Terms

The second step aims at mapping the $n1$ terms of id_1 into the $n2$ terms composing id_2. A term $t_{1,i}$ of id_1 is mapped into a term $t_{2,j}$ of id_2 according to three criterion: the two terms match, the terms have a low string distance, or a semantic relation exists between them, as shown in Algorithm 1. The algorithm uses a function $matching(t_1, t_2, matchType)$ that, given two terms, returns true if the terms match according to the matching criterion specified as third parameter, false otherwise:

1. **Exact**: if the two strings exactly match.

2. **Low string distance**: if the two strings have a *nld* smaller than or equal to a given threshold.

3. **Semantic**: if the two strings have any semantic relation according to the upper ontology *WordNet* [21][2]. In particular, we consider the two terms semantically related if, by querying WordNet, t_1 is synonym, hyponym, hypernym, anyonym, meronym, or holonym of t_2 or, in case none of these relations can be found, *e.g.*, because one of the two terms does not exist in WordNet, the two terms have the same stem according to the Porter [22] stemming.

Algorithm 1 builds a mapping of terms in id_1 into any (not yet mapped) term of id_2, repeatedly traversing id_2 terms, moving from the position of the term of id_1 and using the *exact* matching first, then the *low string distance* matching, and finally the *semantic* matching. The last criterion makes our approach robust by making it able to find relations even for words that are not contained in the WordNet database and–or that have simpler relations, such as one term being the plural or the other.

Finally, our approach excludes terms that are mapped exactly and in the same position in the identifier because we are interested in renamings. For example, in the identifier renaming *getFieldsReferencedIn* → *getFieldReferencesIn*, both identifiers contain *get* and *In* that are exact matches and thus are removed from further consideration. More complicated

[2] `http://wordnet.princeton.edu/`

cases where the exact match is not encountered in a corresponding position will be considered in future work because it may require mapping identifiers into sentences.

3.2.3 Classifying Term Renamings

After terms of id_1 have been mapped to terms of id_2, our approach classifies the renamings at term level, as:

1. **Removed:** terms of id_1 not mapped into any term of id_2 are classified as removed.
2. **Added:** terms of id_2 not mapped into any term of id_1 are classified as added.
3. **Matched:** terms of id_1 mapped into terms of id_2 according to Algorithm 1 with an exact match.
4. **Similar:** terms of id_1 mapped into terms of id_2 according to Algorithm 1 with a semantic relation or a low string distance.

A more sophisticated approach could map sequences of additions and removals into changes, as the Unix *diff* does on source code lines. For example, we treat the renaming of *toStringValue* into *printValue* as the removal of *to* and *String*, and the addition of *print*, while a *diff* would see it as *to String* being changed into *print*. However, such a mapping is out of scope of this paper and we plan to incorporate such a feature in future work on our renaming classifier.

Finally, our approach classifies *similar* terms by:

- determining whether the textual similarity is low or high, according to the *nld* and the 0.40 threshold;
- determining whether the term has changed its grammar form. We use WordNet to determine the grammar form of each term, whenever possible. WordNet classifies (n)ouns, (a)djectives, adve(r)bs, and (v)erbs. In some cases, a term can belong—depending on the context—to more than one grammar form. In this case, WordNet returns a set of grammar forms.
- Determining, again using WordNet, the semantic relation between terms, *i.e.*, synonymy, hyponymy, hypernymy, anyonymy, meronymy, or holonymy, and, if none, whether the two terms have the same stem.

4. EMPIRICAL STUDY

The **goal** of this study is to perform an exploratory analysis of identifier renamings, with the **purpose** of understanding *when* developers rename identifiers, *who* performs such renamings, and *how* identifiers are renamed according to our taxonomy. The **quality focus** is code understandability and maintainability, which, according to previous literature studies, may depend on the quality of identifiers. The **perspective** is mainly of researchers interested in better understanding renaming activities in software project. The **context** consists on a subset of the history of two open-source systems, Eclipse-JDT and Tomcat.

Eclipse-JDT is a set of plug-ins that adds the capabilities of a full-featured Java IDE to the Eclipse[3] platform. Tomcat[4] is an open-source implementation of a servlet container and JavaServer page engine. It evolved over the years to include various features such as load balancing, security managers, connector to Apache (the main project), virtual hosting, management, just to mention a few.

[3] `http://www.eclipse.org/`
[4] `http://tomcat.apache.org/`

Table 1: Characteristics of the analyzed systems.

System	Analyzed period	Files (range)	KLOCs (range)	Files (total)	File revisions	Committers
Eclipse-JDT	2001–2006	2,089–6,949	205–534	5,758	54,571	50
Tomcat	1999–2006	51–1,099	5–315	12,205	46,498	79

Table 1 reports the main characteristics of the analyzed systems: the analyzed periods, the system size ranges in KLOCs, their number of files, the number of files of which we analyzed the history, the total numbers of analyzed file revisions, and the total number of committers. Eclipse is versioned under CVS while Tomcat under SVN.

4.1 Manual Validation of Identifier Mappings

To estimate the precision of our approach in mapping renamed identifiers, we manually validated all identified renamings (after applying type and similarity filters) for Tomcat and some of the renamings of Eclipse-JDT. Out of 885 renamings identified for Tomcat, we observed 161 false positives (18%). We therefore removed them from further analysis, leaving for Tomcat 724 renamings. For Eclipse-JDT, it was not possible to perform a thorough manual validation because we found 4,500 identifier mappings. We measured 17% false positives on a sample of 203 renamings, which corresponds to a 95% confidence level with a 7% confidence interval, thus the percentage is consistent with the imprecision found for Tomcat. We did not remove the false positives from the results reported for Eclipse-JDT, because only a small subset was manually verified.

4.2 Research Questions

We break down our study into two steps: first, we study renamings in general *i.e.*, an activity performed during software evolution and we seek answers to the following two research questions:

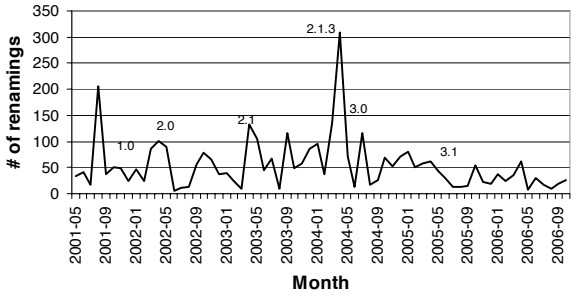
- **RQ1:** *When do identifier renamings happen?* This question investigates how identifier renamings are distributed over time, *i.e.*, if they are spread over time or if there are specific time periods, *e.g.*, in correspondence of some releases, when renamings are performed. We compute and analyze, for each system, the number of renamings performed on a monthly basis.
- **RQ2:** *Who are the developers that mostly perform identifier renamings?* This question investigates whether there are specific developers that performed most of the renamings or all developers performed renamings. We compute the number of renamings performed by each committer and assess who performed more renamings and how many they did.

Then, we study renamings in the context of our taxonomy and answer the following question:

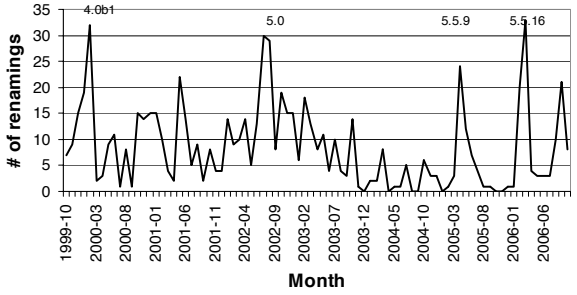
- **RQ3:** *What kind of changes occur in the terms composing renamed identifiers according to our taxonomy?* This research question goes deeply into the nature of renamings by classifying renamings according to our taxonomy and investigating how terms composing identifiers changed. Its focus is in particular on (i) where identifier renamings occur (ii) what kind of semantic

Table 2: Overview of the found identifier renamings.

	Eclipse-JDT	Tomcat
Total # of found renamings	4,500	885
True positives	N/A	724
# (percentage) of renaming changes	2,820 (5%)	575 (1%)
# (percentage) of files affected	1,334 (23%)	396 (3%)
# (percentage) of committers involved	36 (72%)	39 (49%)



(a) Eclipse-JDT



(b) Tomcat

Figure 1: Number of renamings over time.

changes are performed, (iii) the distance between the terms before and after a renaming, and (iv) what kind of grammatical changes are performed.

5. RESULTS

This section reports the results of our empirical study and answers our research questions. Raw and working data sets are available for download on-line[5]. Table 2 shows a summary of the detected renamings. In total, we found, using the similarity and type filters, 4,500 renamings in Eclipse-JDT and 885 in Tomcat. Identifier renamings affected 23% of Eclipse-JDT files and only 3% of Tomcat files.

5.1 RQ1: When Do Renamings Happen?

Figure 1 shows the number of renamings performed in different periods—discretized on a monthly basis—of the analyzed history of the two systems. By looking at the figure and considering the percentages of renamings per file (see second row of Table 2), we observe that renamings are especially concentrated in specific, limited time frames. This observation is not surprising as it is a common behavior for refactorings in general [9]. For Eclipse-JDT, we observe a peak in release 2.1.3. In that period, we found commit notes referring to renaming activities, *e.g.*,: 2004-02-22, *jdesrivieres: "Changes to AST nodes for 1.5"*, or 2004-03-19, *dmegert: "Cleaned up rename of reconcile participant to reconcile listener"*.

For Tomcat, peaks in renaming activities are visible in releases 4.0b1, 5.0 (major release), 5.5.9 and 5.5.16. By looking at the commit notes, in some cases, it is even possible to identify that a major renaming happened in these releases, *e.g.*, 2002-06-08, *remm: "Rename methods getNamingResource -> getNamingResources..."*, 2002-08-26, *patrickl: "Correct method name to match current specification draft"*, or, 2005-04-18/2006-02-07, *fhanik: "Fixed spelling errors"*.

Table 3: Top 10 committers involved in renamings. Values in parentheses indicate the percentages of file changed related to renaming per author.

Eclipse-JDT		Tomcat	
ID	# of renamings	ID	# of renamings
pmulet	792 (3%)	costin	139 (1%)
othomann	269 (3%)	luehe	107 (3%)
jlanneluc	263 (3%)	remm	89 (1%)
maeschli	260 (1%)	fhanik	78 (3%)
jdesrivieres	197 (12%)	craigmcc	51 (1%)
darin	158 (1%)	kinman	29 (1%)
ptff	150 (7%)	markt	27 (0%)
daudel	127 (3%)	amyroh	22 (1%)
maeschlimann	127 (5%)	pier	22 (1%)
kmaetzel	123 (6%)	billbarker	15 (1%)
Total top 10	2,466	Total top 10	579
Total renamings	4,500	Total renamings	724
% renamings top 10	55%	% renamings top 10	80%

5.2 RQ2: Who Performs Renamings?

As shown in Table 2, the numbers of committers involved in renamings was 36 out of 50 (72%) for Eclipse-JDT and 39 out of 84 (49%) for Tomcat. Table 3 shows the list of the top 10 committers involved in identifier renamings, over the 50 Eclipse-JDT committers and 84 Tomcat committers. In Eclipse-JDT, 10 out of 50 committers performed 55% of the renamings while in Tomcat, 10 out of 84 committers performed 80% of renamings. Especially for Tomcat (but to some extent also for Eclipse-JDT), most of the renamings have been performed by a small subset of the committers. As it can be seen in the table, the percentages of commits related to renamings is more or less similar for all committers in the top 10 (*i.e.*, ranging between 1 and 3%), besides a few that have higher percentages, such as *jdesrivieres* in Eclipse-JDT, for whom 12% of commits involved a renaming activity. As illustrated above, *jdesrivieres* was involved in the massive renaming before Eclipse 2.1.3.

5.3 RQ3: What Kind of Renamings Happen?

In the remaining of this Section, we provide insights on term renamings using our taxonomy.

D1: Entity Kind.

Figure 2 shows the renamings classified for different kinds of entities, *i.e.*, the frequencies of renamings for each value of D1. Most of the renamings occur to method or field identi-

[5]http://web.soccerlab.polymtl.ca/ser-repos/public/renaming-data.tgz

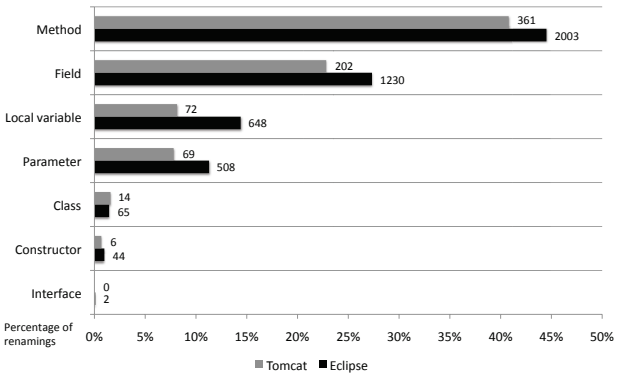


Figure 2: Number of renamings per kind of entity.

fiers: they describe behaviors and properties of classes, thus, very likely, developers rename them to provide more meaningful identifiers. Local variables are less renamed, possibly because their visibility is somewhat limited when compared with method and field identifiers. Finally, it is important to recall that (i) class/interface renamings pertain to renamings of member/inner classes and multiple classes contained within the same file, because other class/interface renamings involved file name changes and thus were not considered in our study; and (ii) constructor renamings occur accordingly. The numbers of class renamings are higher than the numbers of constructor renamings because, for several classes, there are no explicit constructors.

D2: Semantic.

Table 4 describes term renamings based on their semantic (D2), providing some examples for both systems (when available). Surprisingly, we found cases where terms were renamed to their antonym (opposite meaning). Moreover, a manual validation confirmed that renamings to antonyms are more frequent than the frequencies reported here because of the impossibility to detect verb antonyms with WordNet. In our understanding, a renaming to antonym occurs when developers change the way the algorithm that they are implementing works or fix a bug (as it was doing the exact opposite of its expected behavior). For example, an opening bracket was to be matched rather than a closing bracket: *hasClosingBracket* → *hasOpeningBracket*, or an AST navigation should be performed by navigating towards parents and not children *findNextLevelChildrenByElementName* → *findNextLevelParentByElementName*.

We did not find cases for term renamings that hold whole–part relationship. We believe that this observation is partially due to the limited applicability of the WordNet upper ontology to capture whole–part relationships in the specific domain of the two systems *i.e.*, a domain-specific ontology would be needed. In a very few specific cases, our observation can be due to the filtering applied, which excluded renamings where the textual similarity between identifiers was above the fixed threshold. We looked at the results without applying the filter and found one case of renaming towards a meronym: *filename* → *extension*.

Finally, we observe that very few term renamings changed the meaning of a term, about 5% for both systems.

D3: String Distance.

Table 5 shows the number of term renamings for dimension D3, which captures the distance between the original and modified terms. The results correspond to all term renamings, no matter how they are spread over identifiers. We considered additions and removals as having high distances, *e.g.*, in *addPointer* → *Pointer* the term *add* is removed which means that *add* is mapped to an empty string and the corresponding distance will be considered as high. Renamings with low distance are related to prefix/suffix change, fixing typos, or expanding/abbreviating terms.

D4: Grammatical.

Finally, Table 6 shows how term renamings are spread over the different values of D4, *i.e.*, changes of grammar forms. These values were computed among cases where terms underwent a semantic changes. The numbers of grammar form changes among terms in the renamings are extremely low, which we explain as an identifier has been conceived to describe an entity (*i.e.*, by using a noun) or an action (*i.e.*, by using a verb). As we have observed for D2, such identifier can change towards a more precise/less, precise/opposite meaning but it is unlikely that it will change grammar form, except for particular cases.

We observe a relatively high number of "other changes" (347 terms for Eclipse-JDT and 27 for Tomcat). These changes are mainly related to cases where WordNet is not able to uniquely identify the term grammar form: some English terms, depending on the context, can assume different grammar forms: *e.g.*, *monitor* can be both a noun or a verb and *set* can be noun, verb, or adjective. Examples of changes where the term had a multiple grammar classification are (from Eclipse-JDT): *schedule* (noun/verb) → *scheduling* (noun), *dead* (noun/adjective/adverb) → *deadlock* (noun), or *filename* (noun) → *file* (noun/verb).

5.4 Why Do Renamings Happen?

We now discuss some examples of renamings that we found during the manual validation of Tomcat and provide some likely reasons based on our observations.

Formatting: in some cases renaming was likely undertaken for readability or to abide by code conventions (*e.g.*, Camel Case: *appbase* → *appBase*, underscore: *SC_A_SSL_KEYSIZE* → *SC_A_SSL_KEY_SIZE*).

Improving abbreviations: developers seem to pay attention on the abbreviations used for a given term (*e.g.*, *TYPE_CONF_APPLIC* → *TYPE_CONF_ENUM_APPL*).

Never satisfied with a term: renaming sometimes is an iterative process. During our analysis, we encountered multiple renamings for the same declaration (*i.e.*, *list* → *roleList* → *roles*).

Propagation to different packages: when renaming variables that exist in multiple packages, developers do it consistently, *e.g.*, *size* → *capacity* in `WarpTable.java` in two different folders.

Declarations consistent with comments: renaming sometimes happens to make a declaration consistent with its comment (*i.e.*, *list* → *roleList*, where the comment is: "*// Accumulate the user's roles*".

Table 4: Classification of term semantic changes.

Renaming	Eclipse-JDT	Tomcat	Example[6]
add meaning	3,333	357	type → *auth*type (T) resource → visited*Resource* (E)
remove meaning	2,580	326	copy*JAR* → copy (T) f*Type*Binding → fBinding (E)
same meaning	436	42	*committed* → *commited* (T) methods*Buffer* → methods*Buffered* (E)
generalization/specialization relation	24	0	scanCurrent*Position* → scanCurrent*Line* (E) thrownException*Size* → thrownException*Length* (E)
opposite meaning	17	0	findNextLevel*Children*ByElementName → findNextLevel*Parent*ByElementName (E) has*Closing*Bracket → has*Opening*Bracket (E)
whole/part relation	0	0	
unrelated meaning	989	207	create*Contents* → create*Control* (E) get*Cluster*Receiver → get*Channel*Receiver (T)
Total	7,379	932	

(In)consistency in renamings: consistency does not seem to be always a concern. Sometimes, original and renamed terms are different abbreviations for a same word. For example, we observed two field declarations, both of type *int*, that were renamed: *TYP_HOST_ID* → *TYP_CONINIT_HID* and *TYPE_HOST* → *TYP_CONINIT_HST*. Similarly, we observed different expansions of a same word but in different grammatical forms. Two attributes of the same type (*int*) were renamed: *RID_CONN* → *RID_CONNECTION* and *RID_DISC* → *RID_DISCONNECT*.

Table 5: Classification of term string distance.

Distance	Eclipse-JDT	Tomcat	Example[6]
low	1,433	249	isOverridden*Method* → are-Overridden*Methods* (E) statement → stmt (E) parameters → params (E) warining → warning (E) message → msg (T)
high	5,946	683	*is*OverriddenMethod → *are*OverriddenMethods (E)
Total	7,379	932	

6. THREATS TO VALIDITY

To the best of the authors' knowledge, this is the first study of renaming analyzing two large open-source systems. As such, there are several limitations and threats to its validity. There are no threats to *internal validity* nor to *conclusion validity*, our study being an exploratory study.

Construct validity threats are mainly due to limitations and limited precision/recall of the renaming identification approach. In this first exploratory study, we limited ourselves at considering pairs of files that did not change name and location between two versions. We explain our choice with the need for performing an extensive and thorough manual validation of the results and avoiding ambiguity in the decision about class evolution *i.e.*, if indeed a class evolved changing its name and position in the package hierarchy.

Concerning the precision, in Section 4.1, we have discussed performances of our approach, at least on a subset of the extracted renamings. We removed Tomcat false positives (about 18%) from our data set, while we expect for Eclipse-JDT an imprecision of about 17%, according to the manually validated sample. In fact, we manually verified 203 randomly-selected Eclipse-JDT renaming and we found 83% precision (17% imprecision) that corresponds to a 95% confidence level with a 10% confidence interval.

Concerning the recall, we are aware that, as discussed in Section 3.1.3, the adopted filters can filter out some good renamings, *e.g.*, those where the *nld* is higher than the threshold and those where types changed. Also, as stated above, we do not consider file-level renaming, for which an origin analysis would be needed. Thus, our results are a lower bound as there is a percentage of missed renamings and that any improvement will necessarily increase the number of discovered renamings although the ratios between the different categories of renamings may be modified. We applied a graph-matching approach [12] to the entire Eclipse history and all files/classes and we found that, on average, (public) class renamings impacts happen on no more than 1%–5% of the classes (depending on the time window considered). Thus, we believe our results will not be substantially modified once our approach will be integrated with origin analysis and that the achieved accuracy is acceptable considering the linear complexity of the approach.

Furthermore, it would be most likely difficult to apply more precise approaches such as [19] on thousands of file changes for systems of Tomcat or Eclipse-JTD sizes. In fact, [19] has a worst case complexity that is quadratic in the number of system tokens, which is a severe limitation for the sizes of the analyzed systems. Finally, as already noticed, building precise and accurate call graphs with polymorphism resolution for about 100,000 file changes to accurately identify methods renamings would be infeasible in an acceptable amount of time even considering incremental call graphs and data dependencies analysis update.

As for the classification of semantic changes of the terms, our approach fails to classify some renaming due to the presence of abbreviations (or expansions) and typos (the word is not found in the WordNet database), *e.g.*, *message* → *msg* is classified as an unrelated change even if, based on our taxonomy, it should be classified as "same meaning".

External validity threats concern the generalization of our findings. Although the study was performed on a long history (six and eight years) of two medium/large systems, it is desirable to replicate the study on other systems, *e.g.*, developed using different programming languages. Also, with the availability of a renaming identification approach able to avoid the limitations of the used filters, we could obtain different proportions of some renamings.

[6]Renamed terms are in italics.

Table 6: Classification of term grammar changes.

Renaming	Eclipse-JDT	Tomcat	Example[6]
noun to verb	4	0	*editor* → *edit* (E)
noun to adjective	7	0	*qualification*Pattern → *qualified* Pattern (E)
verb to noun	4	2	prepared*Authenticate* → prepared*Credentials* (T)
verb to adjective	5	0	f*Reconcile*Listeners → f*Reconciling*Listeners (E)
adjective to noun	5	0	fLayout*Hierarchical*Action → fShowTest*Hierarchy*Action (E)
adjective to verb	2	0	is*Valid*ClassFile → *validate*ClassFile (E)
adverb to adjective	0	0	
Other changes	347	27	filter*Static* (n;a) → filter*Statics* (n)
No change	230	27	

7. RELATED WORK

This section discusses related work on the role of identifiers in software quality and approaches for refactoring.

7.1 Role of Identifiers in Software Quality

There is quite a consensus among researchers [4, 5, 13, 8] on the role played by identifiers on program comprehension, maintainability, and quality in general. In particular, researchers studied the usefulness of identifiers to recover traceability links [2, 18], measure conceptual cohesion and coupling [20, 23], and, in general, high quality identifiers are considered an asset for source code understandability and maintainability [24, 15, 14].

As suggested in [5], identifiers should be consistent and concise; unfortunately, verifying consistency and conciseness is a difficult task and thus approaches have been developed to identify consistency and conciseness violations via synonyms and homonyms identification [13]. We share the concern expressed in previous studies on identifier quality as a support for various software engineering tasks. However, we are focusing our study on identifier renaming based on a newly proposed taxonomy. We concur with [13] that synonyms play an important role but we also believe that synonyms renamings as well as other phenomena, such as hypernyms, hyponyms, and antonyms, should be investigated as they likely point to program understanding issues.

7.2 Refactoring

Demeyer et *al.* [6] detect object-oriented refactorings based on a set of heuristics defined in terms of changes of object-oriented metrics measuring two successive software versions. The authors validated the approach on several successive versions of three cases studies implemented in Smaltalk.

Dig et *al.* [7] proposed a tool (Refactoring Crawler) for detecting sequence of refactoring actions between consecutive versions of Java systems. Refactoring Crawler identifies seven types of refactoring: rename package, class, and method; pull up and push down method; move method and change method signature. The detection algorithm consists of a fast syntactic analyzer followed by a more computationally intensive semantic analyzer. Refactoring Crawler was applied on various releases of Eclipse UI, Struts, and JHotDraw with accuracy as high as 85%.

Recently Xing et *al.* [25] presented a tool, UMLDiff, and an approach to detect refactorings at design level. UMLDiff works with class diagrams; it inputs two class diagrams and it produces as output an XML design differencing file. Queries on the XML file difference allow to detect simple (*e.g.*, rename class/method/field, pull-up/push-down method/field) and composite refactoring actions (*e.g.*, form template method, replace inheritance with delegation).

Malpohl et *al.* [19] presented a tool, Renaming Detector, for detecting identifier renamings. The tool uses three main components: Parser, Symbol Analyzer, and Differencer. Renaming Detector analyzes each file for extracting identifier declarations and references. Next, it matches the declarations in two versions of a file. In addition, to increase accuracy, types of variables and references are compared for matching the identifiers. For the evaluation purpose, the authors applied the tool on the source code of the tool itself; authors report a 100% precision rate for 77 analyzed file pairs with the computation time around nine minutes.

We share with Refactoring Crawler, UMLDiff and Renaming Detector their general ideas and goal. We also use parsing and differencing technologies though in different combination to attain an approximated, lightweight, robust, and scalable approach. The novelty of our work is a renaming taxonomy directly conceived to better represent renamings on orthogonal dimensions. Furthermore, our approach does not require a compilable system to work; and it detects finer-grain details about renamings, such as the grammatical renaming type or the semantic type.

8. CONCLUSION

This paper presented the first taxonomy for term renamings and reported the first exploratory study investigating and automatically classifying renamings observed in the life span of two real-world systems, Eclipse-JDT and Tomcat. The taxonomy is based on four dimensions, characterizing term renamings orthogonally along four dimensions.

We found a relatively large number of renamings, mostly concentrated in specific time frames and performed by a subset of the committers, sometimes the most active ones. When classifying renamings according to the proposed taxonomy, (i) we found that most of the identified renamings occur on class interfaces (method and field identifiers), likely because their meaning is more important than that of local variables; (ii) we observed term renamings not only towards synonyms but also antonyms and meronyms, which points to the possibility of renaming as a step to correct wrong semantic; (iii) we reported that small string changes are often due to typos or expansions/contractions; and (iv) we found that the grammar forms of terms tend not to change often.

The most frequent renaming changes were adding and removing terms but we cannot draw a conclusion on the impact of such changes on identifiers as, for example, adding and removing terms may lead the changed identifier to be a synonym to the original one. For 5% of the term renamings, meanings were unchanged because synonyms were involved. In few renamings, we also found antonyms (Eclipse-JDT) and whole–part (manually found) relationships. Our results

support the conjecture that renaming is not a one-time activity but has burst of activities, around major releases.

Future work will be devoted to further improve our taxonomy, in particular by studying the combination of term renamings at the identifier-level. We will also merge our approach with a class evolution mapping approach to propose a complete view of renamings. We will possibly integrate more sophisticated heuristics. We will also investigate the feasibility to integrate origin analysis, study term renaming effect on identifier meaning, and include the developers' intention in our taxonomy. Finally, we will investigate the relationship, if any, between the different dimensions of the taxonomy and evolution phenomena, such as class change- or fault-proneness.

9. REFERENCES

[1] S. L. Abebe, S. Haiduc, A. Marcus, P. Tonella, and G. Antoniol. Analyzing the evolution of the source code vocabulary. In *Proceedings of the European Conference on Software Maintenance and Reengineering*, pages 189–198. IEEE CS Press, 2009.

[2] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo. Recovering traceability links between code and documentation. *IEEE Transactions on Software Engineering*, 28(10):970–983, 2002.

[3] G. Canfora, L. Cerulo, and M. Di Penta. Tracking your changes: a language-independent approach. *IEEE Software*, 27(1):50–57, 2009.

[4] B. Caprile and P. Tonella. Restructuring program identifier names. In *Proceedings of the International Conference on Software Maintenance*, pages 97–107. IEEE CS Press, 2000.

[5] F. Deissenbock and M. Pizka. Concise and consistent naming. In *Proceedings of the International Workshop on Program Comprehension*. IEEE CS Press, 2005.

[6] S. Demeyer, S. Ducasse, and O. Nierstrasz. Finding refactorings via change metrics. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 166–177. ACM Press, 2000.

[7] D. Dig, C. Comertoglu, D. Marinov, and R. E. Johnson. Automated detection of refactorings in evolving components. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 404–428. Springer, 2006.

[8] E. Enslen, E. Hill, L. L. Pollock, and K. Vijay-Shanker. Mining source code to automatically split identifiers for software analysis. In *Proceedings of the International Working Conference on Mining Software Repositories*, pages 71–80, 2009.

[9] M. Fowler. *Refactoring: Improving the design of existing code*. Addison-Wesley, 1999.

[10] M. Godfrey and Q. Tu. Tracking structural evolution using origin analysis. In *Proceedings of the International Workshop on Principles of Software Evolution*, pages 117–119. ACM Press, 2002.

[11] S. Haiduc and A. Marcus. On the use of domain terms in source code. In *Proceedings of the International Conference on Program Comprehension*, pages 113–122. IEEE CS Press, 2008.

[12] S. Kpodjedo, F. Ricca, P. Galinier, and G. Antoniol. Recovering the evolution stable part using an ECGM algorithm: Is there a tunnel in Mozilla? In *Proceedings of the European Conference on Software Maintenance and Reengineering*, pages 179–188. IEEE CS Press, 2009.

[13] D. Lawrie, H. Feild, and D. Binkley. Syntactic identifier conciseness and consistency. In *Proceedings of the International Workshop on Source Code Analysis and Manipulation*, pages 139–148. IEEE CS Press, 2006.

[14] D. Lawrie, C. Morrell, H. Feild, and D. Binkley. What's in a name? a study of identifiers. In *Proceedings of the International Conference on Program Comprehension*, pages 3–12. IEEE CS Press, 2006.

[15] D. Lawrie, C. Morrell, H. Feild, and D. Binkley. Effective identifier names for comprehension and memory. *Innovations in Systems and Software Engineering*, 3(4):303–318, 2007.

[16] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Cybernetics and Control Theory*, 10(8):707–710, 1966.

[17] N. Madani, L. Guerrouj, M. Di Penta, Y.-G. Guéhéneuc, and G. Antoniol. Recognizing words from source code identifiers using speech recognition techniques. In *Proceedings of the European Conference on Software Maintenance and Reengineering*, pages 68–77. IEEE CS Press, 2010.

[18] J. I. Maletic, G. Antoniol, J. Cleland-Huang, and J. H. Hayes. In *International Workshop on Traceability in Emerging Forms of Software Engineering*, page 462. ACM Press, 2005.

[19] G. Malpohl, J. J. Hunt, and W. F. Tichy. Renaming detection. In *Proceedings of the International Conference Automated Software Engineering*, pages 73–80. IEEE CS Press, 2000.

[20] A. Marcus, D. Poshyvanyk, and R. Ferenc. Using the conceptual cohesion of classes for fault prediction in object-oriented systems. *IEEE Transactions on Software Engineering*, 34(2):287–300, 2008.

[21] G. A. Miller. Wordnet: A lexical database for english. *Communications of the ACM*, 38(11):39–41, 1995.

[22] M. F. Porter. An algorithm for suffix stripping. *Program*, 14(3):130–137, 1980.

[23] D. Poshyvanyk and A. Marcus. The conceptual coupling metrics for object-oriented systems. In *Proceedings of the International Conference on Software Maintenance*, pages 469–478. IEEE CS Press, 2006.

[24] A. Takang, P. A. Grubb, and R. D. Macredie. The effects of comments and identifier names on program comprehensibility: an experiential study. *Journal of Program Languages*, 4(3):143–167, 1996.

[25] Z. Xing and E. Stroulia. Refactoring detection based on UMLDiff change-facts queries. In *Proceedings of Working Conference on Reverse Engineering*, pages 263–274. IEEE CS Press, 2006.