REPENT: Analyzing the Nature of Identifier Renamings

Venera Arnaoudova[1], Laleh M. Eshkevari[1], Massimiliano Di Penta[2],
Rocco Oliveto[3], Giuliano Antoniol[1], Yann-Gaël Guéhéneuc[1]
[1]École Polytechnique de Montréal, Québec, Canada
[2]University of Sannio, Benevento, Italy
[3]University of Molise, Pesche (IS), Italy
venera.arnaoudova@polymtl.ca, laleh.eshkevari@polymtl.ca, dipenta@unisannio.it,
rocco.oliveto@unimol.it, antoniol@ieee.org, yann-gael.gueheneuc@polymtl.ca

Abstract—
Source code lexicon plays a paramount role in software quality: poor lexicon can lead to poor comprehensibility and even increase software fault-proneness. For this reason, renaming a program entity, *i.e.*, altering the entity identifier, is an important activity during software evolution. Developers rename when they feel that the name of an entity is not (anymore) consistent with its functionality, or when such a name may be misleading. A survey that we performed with 71 developers suggests that 39% perform renaming from a few times per week to almost every day and that 92% of the participants consider that renaming is not straightforward. However, despite the cost that is associated with renaming, renamings are seldom if ever documented—for example, less than 1% of the renamings in the five programs that we studied. This explains why participants largely agree on the usefulness of automatically documenting renamings. In this paper we propose REPENT (REANAMING PROGRAM ENTITIES), an approach to automatically document—detect and classify—identifier renamings in source code. REPENT detects renamings based on a combination of source code differencing and data flow analyses. Using a set of natural language tools, REPENT classifies renamings into the different dimensions of a taxonomy that we defined. Using the documented renamings, developers will be able to, for example, look up methods that are part of the public API (as they impact client applications), or look for inconsistencies between the name and the implementation of an entity that underwent a high risk renaming (*e.g.*, towards the opposite meaning). We evaluate the accuracy and completeness of REPENT on the evolution history of five open-source Java programs. The study indicates a precision of 88% and a recall of 92%. In addition, we report an exploratory study investigating and discussing how identifiers are renamed in the five programs, according to our taxonomy.

Index Terms—Identifier renaming, Refactoring, Program comprehension, Mining software repositories, Empirical Study

1 Introduction

When the source code of a program evolves [36], its identifiers evolve too [2]. Thus identifier renaming, *i.e.*, the activity during which an entity—*e.g.*, a local variable, a method, or a class—changes its name, has a paramount importance in software evolution[1]. Developers rename when they feel that the name of an entity is not (anymore) consistent with its functionality or when such a name may easily create understanding problems.

We surveyed 71 developers of industrial and open-source systems about their renaming habits. We observe that renaming is tangled with many development activities—most of the participants perform renaming while performing other refactorings (90% of surveyed developers); changing or adding functionality (89% and 65% respectively); understanding existing code (51%); or fixing bugs (42%). Renaming is an activity that 39% of participants perform from a few times per week to almost every day and 46% perform a few times per month. Participants mainly use automatic tools to perform renaming (72%), although 20% say that they rename manually and 8% do it in both ways. However, although tool support is available, 92% of the participants consider renaming not straightforward and only 24% think that in most cases renaming has no cost. Indeed, the largest fraction of the surveyed developers (67%) believes that the cost of renaming depends on the particular case and that it requires time and effort. For example, participants underline that renaming identifiers that belong to non-local scope may break backward compatibility, increase integration cost, or impair program understanding for those already familiar with the old name.

However, despite the renaming cost, risks, and implications for program understanding, only a small percentage of the renamings is actually documented—1% of the renamings in the five programs that we studied. This explains why a high number of partic-

[1] Renaming is *per se* considered a refactoring activity [21]. In this paper, we focus only on how developers change the source code lexicon rather than on how the source code is restructured.

ipants (52%) consider the automatic documentation of renamings useful; one of the surveyed developers explains that *"if there were an easy way to look renaming up, it would potentially be informative when backtracking for problems, or even just trying to understand someone else's code. You can often learn a lot about what something does by looking at how other people disagree regarding what it does."* The developer echoes George Santayana: *"Those who cannot remember the past are condemned to repeat it"* [48]. We share their point of view and argue that documenting renamings is important to track changes in vocabulary and to create traceability links between entities over time.

This paper describes REPENT (RENAMING PROGRAM ENTITIES), an approach to detect identifier renamings across different versions of a program and to automatically classify renamings according to a taxonomy that indicates (i) what kind of identifier was renamed (*e.g.*, class name), (ii) whether one or more terms composing the identifier were added/removed/changed (*e.g.*, a term is added when renaming `files` to `srcFiles`[2]), and (iii) how terms were changed with respect to their semantics (*e.g.*, towards opposite meaning when renaming `disable` to `enable`) and grammar (*e.g.*, from adjective to noun when renaming `localDeclaration` to `location`).

REPENT combines tracking of changed Java source code lines with data-flow analysis on programming entities. It first identifies changed source code lines using a lightweight file differencing tool from which it extracts declared entities and maps them across versions to identify candidate renamings. Then, REPENT applies def-use analysis for the entities participating in candidate renamings to further reduce false positives. Finally, REPENT uses WordNet[3] [43] and the Stanford Part-of-Speech Analyzer [53] to classify the detected renamings according to our taxonomy.

We argue that integrating REPENT into existing version control systems or into Integrated Development Environments (IDE) will allow developers to browse for renamings as they feel the need. For example, developers will be able to look up methods that are part of the public API as those renamings must be documented in the release notes for client applications. Developers will also be able to look for inconsistencies such as (i) methods whose name changed from singular to plural but whose return type did not change accordingly (*e.g.*, is not a collection) and (ii) fields whose name changed towards the opposite meaning, but whose functionality or documentation are not consistent.

Using REPENT, we document—*i.e.*, detect and classify—the renamings of five Java programs (ArgoUML, dnsjava, Eclipse-JDT, JBoss, and Tomcat) over several years of evolution history. We show that REPENT accurately (precision of 88% and a recall of 92%) detects renamings throughout the evolution history of large projects. We discuss interesting examples of renamings in those programs as well as possible reasons for some of the renamings.

In the rest of this section we define the necessary terminology (Section 1.1) and provide an overview of the structure of this paper (Section 1.2).

1.1 Definition of Renaming

Identifier renaming is the activity during which an entity changes its name from an **old name** to a **new name**. Renaming alters **entities names** (*e.g.*, classes, methods). We make no distinction between **name** and **identifier**; we use the two words interchangeably. Identifiers (*e.g.*, `cpuMaxClock`) are composed of **terms**, where each **term** is a dictionary word, an abbreviation, or an acronym (*e.g.*, `cpu`, `Max`, and `Clock`).

Renaming an identifier implies adding, removing, changing, or reordering terms of the old name. Thus, an identifier renaming can be seen as being composed of a set of **term renamings**, in which **old terms** are changed to **new terms**.

1.2 Paper Structure

Section 2 motivates this research and reports results of the survey that we performed. Section 3 describes the taxonomy we defined to classify identifier renamings. Section 4 describes the proposed approach to (i) detect identifier renamings and (ii) classify them according to our taxonomy. Section 5 aims at empirically evaluating the accuracy and completeness of REPENT renaming detection. Section 6 reports and discusses the results of the exploratory study[4]. Section 7 discusses threats that could affect the validity of our empirical studies. Section 8 discusses the related literature, while Section 9 concludes the paper and outlines directions for future work.

2 A DEVELOPERS' SURVEY ON IDENTIFIER RENAMING

This section motivates the need for automatic renaming detection and classification. For this purpose, we designed an online survey to understand the importance of renaming, *i.e.*, to what extent developers of

[2] All examples, unless stated otherwise, are taken from the five programs that we studied and are available in an online index at http://ser.soccerlab.polymtl.ca/ser-repos/public/tr-data/repent-index/repent-index.html.

[3] http://wordnet.princeton.edu

[4] The working data set for the study reported in this paper is available online at http://ser.soccerlab.polymtl.ca/ser-repos/public/tr-data/repent-data.tgz.

open-source/industrial projects perform identifier renaming, under what circumstances, and whether they believe that identifier renaming requires automatic documentation. We invited 739 developers, via e-mail using a convenience sampling [23], involving (i) original developers of the five Java programs that we study in this paper and (ii) other developers from the industry and open-source communities. 71 developers responded to the survey resulting in a response rate close to 10% as expected [23]. Although we profile survey participants based on their background, their identity is kept anonymous for confidentiality purposes. Figures with detailed results are reported in Appendix A.

In the following we summarize the results of the survey and we illustrate them with comments from the participants. We complement the survey output with examples that we collected from online discussions of the analyzed programs (issue reports, mailing lists, and commit notes).

How often do developers rename?: Renaming is an activity that participants perform from almost every day (21%), a few times per week (18%), a few times per month (46%), to once per month (14%). A developer commented: *"There's a balance to be struck: - identifiers are communication, and as the code is refactored it is critical that identifiers continue to correctly describe their purpose - changing identifiers tends to break APIs, and sometimes they're used for unintended purposes, over-frequent change is not good."*

Is renaming straightforward?: When we asked participants whether renaming has a cost, only 8% answered that renaming is straightforward. 24% of participants think that in most cases renaming has no cost, often due to the availability of automatic tool support. Indeed the majority of participants (72%) use automatic tool support to perform renaming, although 20% rename manually and 8% use a mix of both, *i.e.*, rename manually and automatically. 32% of participants believe that the cost of renaming depends on the particular case: *"Renaming identifiers that belong to non-local context (e.g., public or protected methods) has a potentially massive cost associated with breaking the interfaces between components. Otherwise it is typically a rather cheap and non-disruptive exercise that may have end benefit of more readable and consistent code. Another element of cost and risk is when the identifiers are being bound to at runtime only (e.g., when classes are loaded by name or methods are bound by name). It is not always easy to trace all such use cases in a large system."* Indeed, renaming an entity that is part of a public API of a program has a higher cost as it breaks backward compatibility and increases the integration cost of the program in client programs. 10% of participants believe that in most cases renaming has a cost, and finally 25% answer that renaming defiantly requires time and effort. Another example where renaming has a cost is when the team uses code reviews, as developers must schedule a code review and justify their decision. A developer indicated that code reviews impact the frequency of renaming *"because you appear negatively to the boss when asking for a review on a 'too minor improvement"'*. The cost of renaming also includes the cost of finding a proper name and assuring that the new name reflects the purpose of the entity in all scenarios that it is used. Quotes like *"I have the feeling that your method name is not good [..]"* for method `getBufferForWrite` in an Eclipse issue report (issue #332248) indicates that, indeed, developers spend time understanding the rationale behind names that are chosen by other teammates.

Already postponed a renaming?: It also appears that, although necessary, some renamings are delayed. After discussing the difference between the term "delete" and "remove", an ArgoUML developer concluded that: *"[..] maybe I shall rename these after next release"* (issue #2938). We asked participants to share reasons for which they recall having decided not to rename an entity. 52% recall the reason to be the potential impact on other systems. A developer explains: *"As a middleware developer, providing a stable API is paramount for clients. There are numerous cases where we would not rename a class or method despite an obviously better name being proposed, in order to minimize the cost of integrating new versions."* 35% recall that the renaming was too risky, *i.e.*, it might have introduced a bug—a developer recalls: *"I encountered a problem when my colleague wrote Java code which uses reflection. I avoided renaming some classes/methods which will be inspected by the reflection, since doing so can introduce unpredictable bugs."* 25% of participants answered that the high impact of the renaming on the system was the show-stopper and finally, 25% recall deciding not to rename because of the high effort required: *"I'm not touching poorly-worded APIs which are shared across multiple projects - the cost of the change does not justify it [..]."* Participants also shared that the impact on other developers is sometimes decisive: *"If too many people in the company know a thing by name X it's sometimes better to keep it even when name Y is more descriptive."* Other factors impacting the decision to undertake a renaming are insufficient domain knowledge (85% of participants), code ownership (79%), and close deadline (76%).

How can REPENT help in such a context?: Detecting and classifying renamings with REPENT—for example generating parts of commit notes when renaming occurs—can be used by developers while backtracking bugs or understanding changes of program entities. REPENT allows developers to differentiate and thus document and retrieve all or only certain types of renamings—*e.g.*, renamings towards opposite meaning as they deserve more attention and can be flagged to make sure that they reflect the developer’s intentions. The documentation of renamings is also useful as a starting point for documenting API changes in release notes. Last but not least REPENT can reduce the unnecessary cost of some renamings by informing developers about names that were already changed in the past.

We asked participants whether they consider useful automatically documenting renaming and 52% of them were positive. A developer elaborates: *“It depends on how this was implemented, but if it were field-level history, e.g., like svn records history for a file, then I’m all for it [..]”*; *“Tracking changes to public api is imperative in large fast moving teams.”*

3 Identifier Renaming Taxonomy

REPENT classifies renamings along the dimensions of a taxonomy that extends and refines the taxonomy proposed in our previous work [19]. We built the taxonomy based on a grounded-theory approach [22], [50] considering dimensions that we believe apply to source code identifiers and the terms that compose them. Specifically, we built the taxonomy by looking at identifier renamings, which we manually validated in our previous work [19], and grouping them into categories. The manual analysis required multiple iterations in order to converge and to consider all the dimensions of the proposed taxonomy.

The taxonomy comprises four dimensions, namely *entity kinds*, *forms of renaming*, *semantic changes*, and *grammar changes*. The first dimension distinguishes renamings based on the programming paradigm, whereas the last three dimensions distinguish renamings based on different natural language aspects. A summary of REPENT taxonomy is reported in Table 1. The dimensions are orthogonal and thus each renaming will be classified in each dimension of the taxonomy. However, there are implicit relations between levels of the different dimensions. For example, classifying an identifier renaming in *form of renaming* as *formatting only* implies that in *semantic change* and *grammar change* it will be classified as *none*. Concretely, in the field renaming `invParamsPtr` → `invalidParamReferencesPtr` the term `inv` is expanded to become `invalid`; `Params` changed to `Param`; `Reference` was added; and `Ptr` stayed unchanged (see Fig. 1). According to our taxonomy, this renaming will be classified as follows:

- **Entity kind:** *Field*,
- **Form of renaming:** *Complex* as two terms are changed, and one term is added,

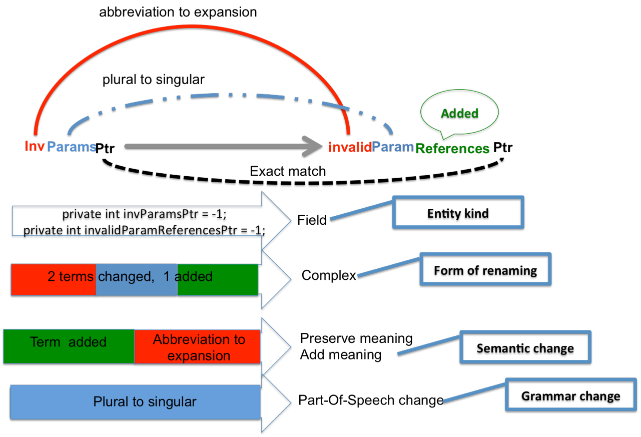


Fig. 1. Example of classifying a renaming based on the proposed taxonomy.

TABLE 1
Summary of the identifier renaming taxonomy

Entity kinds	Package Type Field Constructor Method/Getter/Setter Parameter Local Variable	
Forms of renaming	Simple Complex Formatting only Term reordering	
Semantic changes	Preserve meaning	Synonym Synonym phrase Spelling error correction/introduction Expansion Abbreviation
	Change in meaning	Opposite Opposite phrase Whole-part Whole-part phrase Unrelated
	Narrow meaning	Specialization Specialization phrase
	Broaden meaning	Generalization Generalization phrase
	Add meaning	
	Remove meaning	
	None	
Grammar changes	Part of speech change	Singular/Plural Verb conjugation Other
	None	

- **Semantic change:** *Preserve meaning* as the term renaming `inv` → `invalid` is an expansion, and *add meaning* as the term `Reference` is added,
- **Grammar change:** *Part of speech change* as the term renaming `Params` → `Param` implies a change from plural to singular.

In the rest of this section we describe the different dimensions of the taxonomy.

3.1 Entity Kinds

The first dimension of the taxonomy concerns the kind of the renamed entity, *i.e.*, whether the renamed entity is a package, type (*i.e.*, class, interface, or enumeration), field (*i.e.*, class variables, instance variables, constants, or enumeration constants), constructor, method, getter (*i.e.*, field accessor), setter (*i.e.*, field modifier), parameter, or local variable. We distinguish getters and setters from other methods using naming conventions, *i.e.*, the method name must start with the keywords "get" ("is" for boolean return type) or "set" and of a field name.

3.2 Forms of Renaming

The second dimension provides a classification of the renaming to determine whether one or more terms were changed, whether the renaming was solely related to text formatting, or whether it consisted in term reordering. REPENT considers the following forms.

Simple: Those are renamings where one term has been changed, *e.g.*, `predeclareStatements` → `predeclare` where the term `Statements` has been removed and `override` → `overriding` where the only term composing the old name has been renamed.

Complex: Those are renamings where more than one composing terms have been renamed, *e.g.*, `IsAssignmentWithNoEffectMASK` → `AssignmentHasNoEffect`.

Formatting only: Those are renamings where no term renaming occurs, but rather changing letter cases or adding/removing term separators. Examples include `getJRMPPort` → `getJrmpPort` and `JavaExtension` → `JAVA_EXTENSION`.

Term reordering: Those renamings consist in exchanging the position of terms composing the old identifier. Examples: `setDelaySocketClose` → `setSocketCloseDelay` and `pojoNoInterfacesIntro` → `noInterfacesPOJOIntro`.

3.3 Semantic Changes

The semantic changes dimension concerns changes (or not) in the meaning of the identifier due to the addition/removal of terms to/from the identifier or due to changing one or more terms with terms having different (or same) meaning. As a result, identifiers change while (i) preserving their meaning, (ii) changing their meaning, (iii) adding meaning, or (iv) removing meaning. There is no semantic change when only the format or order of terms change.

3.3.1 Preserve Meaning

Renamings falling in this category preserve the meaning of the identifier.

Synonym: The old and new terms have the same meaning (according to a given ontology). For example, in the renaming `isPotentialMatch` → `isPossibleMatch`, the two terms are synonyms.

Synonym phrase: One or more terms in the old identifier are renamed to one or more terms in the new identifier while preserving the meaning. Example: `javadocNotVisibleReference` → `javadocHiddenReference` where the renamed terms (`visible` and `hidden`) are antonyms and one of them is negated (`visible` and `not visible`).

Spelling error: Examples of correction and introduction of spelling errors are `actionMesasage` → `actionMessage` and `sourceField` → `fiieldInfo` respectively. While one can easily understand the rationale of spelling error correction, spelling error introduction can happen as a side effect of a renaming.

Expansion: A renaming towards expansion occurs when a term—often not belonging to the English dictionary—is expanded into a (longer) term, often belonging to the English dictionary: `setAuthMechanism` → `setAuthenticationMechanism` and `collab` → `collaboration`.

Abbreviation: A renaming towards abbreviation is the opposite of a renaming towards *expansion* and occurs when a term—often belonging to the English dictionary—is contracted into a shorter term, often not belonging to the English dictionary: `packageName` → `pkgName` and `operationDesc` → `opDesc`.

3.3.2 Change in Meaning

Renamings falling in this category include cases in which the renaming changes the meaning of the old identifier.

Opposite: The new term has the opposite meaning of the old term (antonym), *e.g.*, `disableLookups` → `enableLookups`.

Opposite phrase: One or more terms in the old identifier are renamed to one or more terms in the new identifier with an opposite meaning: `isNotPrimitiveType` → `isPrimitiveType` where the term `Primitive` is negated. Also, it can happen that a term is replaced by a synonym and is negated: `isNonModifiableContainer` → `canUpdateContainer`.

Whole-part: The new and the old terms hold a whole-part relation (holonym/meronym); respective examples are `Point` → `Line` (fictitious example) and `body` → `node`.

Whole-part phrase: When more than one whole-part relation exists in the renamed identifiers. An example in this category is `Path` → `FileAndDirectory` (fictitious example[5]).

Unrelated: The old and new terms have unrelated meanings. It is the case for the terms `expression` and `script` in the identifier renaming `expressionModel` → `scriptModel`.

3.3.3 Narrow Meaning

Specialization: The meaning of the old identifier is narrowed when a term is renamed to its hyponym, *e.g.*, `thrownExceptionSize` → `boundExceptionLength`, where the new term `Length` is a hyponym of the old term `Size`.

Specialization phrase: Adding a term that specifies another term narrows the meaning of the old identifier: `item` → `todoItem` and `type` → `authType`. To do so, we consider nouns and adjectives modifiers (as specifiers) added before a term because, based on our qualitative analysis done using grounded theory, these are the most common modifiers that developers use to specify other terms.

3.3.4 Broaden Meaning

Generalization: Opposite to *specialization*, here the old and new terms hold a generalization relation, *i.e.*, the new term is a hypernym of the old term, *e.g.*, `getAccessRestriction` → `getAccessRuleSet`, where the term `Rule` is a hypernym of term `Restriction`.

Generalization phrase: Opposite to *specialization phrase*, here a specifying term is removed. Examples: `eventName` → `name` and `getInitialRepetitions` → `getRepetitions`. As in specialization phrase, we consider nouns and adjectives as modifiers.

3.3.5 Add Meaning

Add meaning renamings happen when an identifier is renamed by adding one or more terms and such a change does not fall into any of the cases discussed above, *i.e.*, in which the meaning is kept or changed. In other words, the added terms add meaning to the identifier, rather than changing (*e.g.*, generalizing, specializing, or negating) the current meaning: `_delete` → `deletePossible` and `flags` → `typeAndFlags`.

3.3.6 Remove Meaning

Remove meaning renamings happen when an identifier is renamed by removing one or more terms and, again, the change does not fall into any of the above cases. That is, the term removal also removes meaning from the identifier. Examples: `includeRule` → `rule` and `removedPackagePath` → `packagePath`.

3.3.7 None

No change in any of the terms in the identifier implies no semantic change, *i.e.*, the *semantic change* will be *none*. This is when the change is only in letter cases or adding/removing term separators.

3.4 Grammar Changes

The grammar changes dimension concerns changes in the part of speech of terms. We further classify part of speech changes into verb conjugation changes, singular to plural changes (and vice versa), or other (*e.g.*, change from noun to adverb).

3.4.1 Part of Speech Change

Those renamings occur when the part of speech of any term composing the old identifier changes. We consider the part of speech set contained in the Penn Treebank Tagset [42]. Thus, a grammar change occurs when an adjective is changed to a verb, as in `getUpdatedSize` → `updateFigGroupSize`. We further focus our attention on nouns and verbs as, to the best of our knowledge, they represent the most critical part of identifiers. Specifically, nouns are shown to be the most important in terms of meaning [11], whereas verbs are used in naming methods and thus changes of verbs can imply changes in functionality [1], [10].

3.4.2 None

There is no grammar change when the modified terms' part of speech remains the same. For example, renaming method `isPotentialMatch` to `isPossibleMatch` does not imply any grammar change as both terms, `Potential` and `Possible`, are tagged as adjective. Moreover, renamings classified as *formatting only* in the *forms of renaming* will also be classified as *none* in the *grammar change*, *e.g.*, `getJRMPPort` → `getJrmpPort`.

4 Renaming Detection and Classification

Fig. 2 shows the processing steps of REPENT at a high level of detail to outline the renaming detection

[5] The example is fictitious as REPENT did not classify any of the renamings detected in the five programs as *whole-part phrase*. However, for the sake of completeness and because results may be different for different programs we decided to keep this level of the taxonomy.

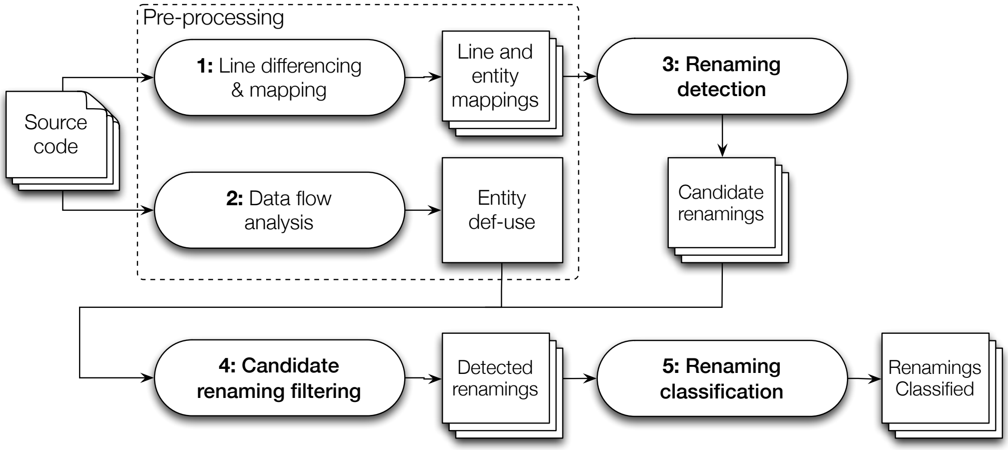


Fig. 2. REPENT: Renaming detection and classification process.

and classification processes. REPENT first detects a set of candidate renamings by means of file context *diff* and filters out false renamings using def-uses pairs, textual analysis, and heuristics. Finally, REPENT classifies renamings along the taxonomy dimensions.

4.1 Renaming Detection

The first step of the process, step 1 in Fig. 2, is detailed in Fig. 3. In this step, REPENT compares two source files by applying a line differencing algorithm, the Unix context diff algorithm, which produces as output a set of line mappings. REPENT uses the mapped source code lines to compare and map entities declared into mapped lines and identify candidate renamings.

The comparison of source files is performed between two consecutive versions of a file or, in case of renamed files, between the file with the old name and the one with the new name. To identify file renamings, REPENT first builds the list of candidate file renamings consisting of (i) all possible couples formed by one file removed in the change set and one file added in the same change set and (ii) explicit renamings in the versioning system (SVN only). For CVS a change set is computed by grouping files based on the commit time stamp (less than 200 seconds between commits belonging to the same change set), commit note, and committer name [58]. Then, REPENT evaluates each couple and selects the best option if the difference between the two files is reasonably low. Specifically, we use the Unix *diff* algorithm to compare the number of changed lines between two files and we consider them as a file renaming if the difference does not exceed a relative threshold of 60%. The value for the threshold is estimated based on the central tendency of explicitly renamed files as logged by the versioning file system.

The output of the comparison between two files is a mapping between lines of the old and new files; four cases must be considered:

1) **One-to-one line mapping:** one line of the old file is mapped onto one line of the new file. Fig. 4 shows an example where line 12 is mapped onto line 14. These two mapped lines are completely unrelated. Indeed, this is a case where the line mapping fails to map the lines correctly. To overcome this limitation of line mapping, we perform a cross validation step (see Section 4.2.4).
2) **One-to-many line mapping:** one line of the old file is mapped onto multiple lines of the new file.
3) **Many-to-one line mapping:** multiple lines of the old file are mapped onto one line of the new file.
4) **Many-to-many line mapping:** multiple lines of the old file are mapped onto multiple lines of the new file. Fig. 5 shows an example where lines 203 to 206 of the old file are mapped onto lines 203 to 206 in the new file.

Once REPENT creates lines mappings, it maps entities that are declared into mapped lines, *i.e.,* it creates entities mappings. To this end, REPENT first parses source code, creates an Abstract Syntax Tree (AST) using Eclipse Java Development Tools (JDT), and identifies the line numbers of all declared entities. Next, given the line mappings and the entities declarations for each source code line, REPENT creates an entity mapping. Again, four cases are possible:

1) **One-to-one entity mapping:** there is exactly one entity of a same kind that is declared in the old line(s) (*e.g.,* local variable) as well as in the new line(s). In this case, the old entity is mapped onto the new entity.
2) **One-to-many entity mapping:** there is only one

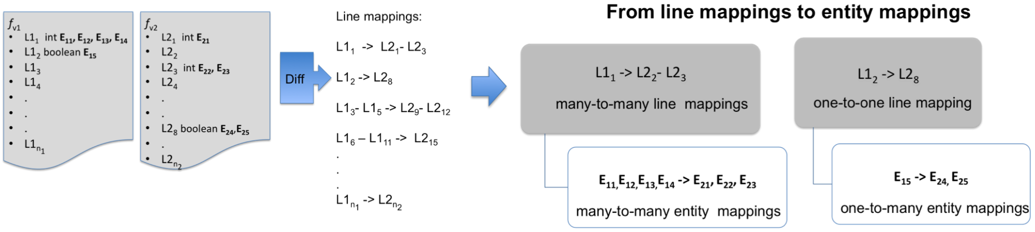
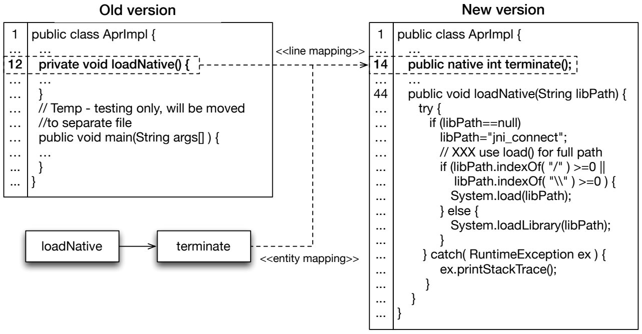

f_{v1}
$L1_1$ int $E_{11}, E_{12}, E_{13}, E_{14}$
$L1_2$ boolean E_{15}
$L1_3$
$L1_4$
$L1_{n_1}$
f_{v2}
$L2_1$ int E_{21}
$L2_2$
$L2_3$ int E_{22}, E_{23}
$L2_4$
$L2_8$ boolean E_{24}, E_{25}
$L2_{n_2}$
Diff
Line mappings:
$L1_1$ -> $L2_1$- $L2_3$
$L1_2$ -> $L2_8$
$L1_3$- $L1_5$ -> $L2_9$- $L2_{12}$
$L1_6$ – $L1_{11}$ -> $L2_{15}$
$L1_{n_1}$ -> $L2_{n_2}$
From line mappings to entity mappings
$L1_1$ -> $L2_2$- $L2_3$
many-to-many line mappings
$L1_2$ -> $L2_8$
one-to-one line mapping
$E_{11}, E_{12}, E_{13}, E_{14}$ -> E_{21}, E_{22}, E_{23}
many-to-many entity mappings
E_{15} -> E_{24}, E_{25}
one-to-many entity mappings

Fig. 3. Details of the renaming detection process.

entity declared in the old line(s), while there are many entities declared in the new line(s). In this case, the old entity is mapped onto several new entities.

3) **Many-to-one entity mapping:** this is the converse of the previous case, *i.e.*, several old entities are mapped onto one new entity.
4) **Many-to-many entity mapping:** there are several entities declared in the old line(s) and in the new line(s). In this case, the old entities are mapped onto the several new entities.

The entity mapping computed at this (early) stage has to be considered as possible mappings and thus as candidates renamings.

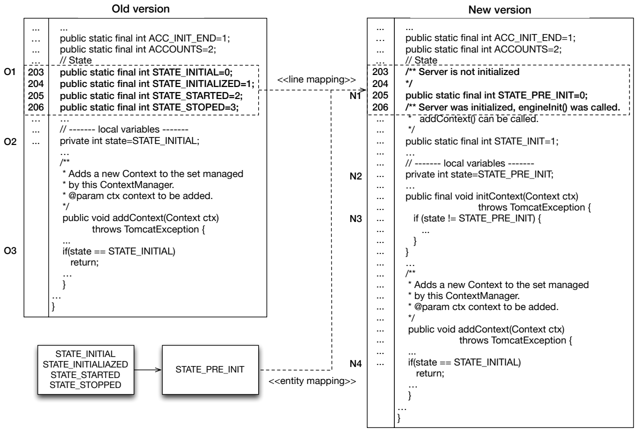


```
1   public class AprImpl {
...   ...
12    private void loadNative() {
...   ...
...   }
...   // Temp - testing only, will be moved
...   //to separate file
...   public void main(String args[] ) {
...   ...
...   }
... }
```

```
1   public class AprImpl {
...   ...
14    public native int terminate();
...   ...
44    public void loadNative(String libPath) {
...     try {
...       if (libPath==null)
...         libPath="jni_connect";
...         // XXX use load() for full path
...         if (libPath.indexOf( "/" ) >=0 ||
...            libPath.indexOf( "\\" ) >=0 ) {
...           System.load(libPath);
...         } else {
...           System.loadLibrary(libPath);
...         }
...     } catch( RuntimeException ex ) {
...         ex.printStackTrace();
...       }
...     }
... }
```

Fig. 4. Example of one-to-one entity mapping.

Fig. 4 shows an example of a one-to-one line mapping corresponding to a one-to-one entity mapping in Tomcat. The developer added a method `terminate` at line 14 of the new file. The line mapping algorithm maps line 12 in the old file—containing the declaration of method `loadNative`—onto line 14 of the new file. REPENT discovers only one entity declared in the old file, as well as in the new one and builds the mapping, *i.e.*, candidate renaming, `loadNative` to `terminate`[6].



```
...
public static final int ACC_INIT_END=1;
public static final int ACCOUNTS=2;
// State
public static final int STATE_INITIAL=0;
public static final int STATE_INITIALIZED=1;
public static final int STATE_STARTED=2;
public static final int STATE_STOPED=3;
...
// ------- local variables -------
private int state=STATE_INITIAL;
...
/**
 * Adds a new Context to the set managed
 * by this ContextManager.
 * @param ctx context to be added.
 */
 public void addContext(Context ctx)
           throws TomcatException {
 ...
 if(state == STATE_INITIAL)
   return;
 ...
 }
...
}
```

```
...
public static final int ACC_INIT_END=1;
public static final int ACCOUNTS=2;
// State
/** Server is not initialized
*/
public static final int STATE_PRE_INIT=0;
/** Server was initialized, engineInit() was called.
 *   addContext() can be called.
 */
public static final int STATE_INIT=1;
...
// ------- local variables -------
private int state=STATE_PRE_INIT;
...
public final void initContext(Context ctx)
                     throws TomcatException {
  if (state != STATE_PRE_INIT) {
     ...
  }
}
...
/**
 * Adds a new Context to the set managed
 * by this ContextManager.
 * @param ctx context to be added.
 */
 public void addContext(Context ctx)
                   throws TomcatException {
 ...
 if(state == STATE_INITIAL)
   return;
 ...
 }
...
}
```

Fig. 5. Example of many-to-one entity mapping.

Fig. 5 shows a less trivial example where many entities are declared into the mapped lines (*i.e.*, many-to-many line mapping and many-to-one entity mapping). Thus, in this case, four entities (`STATE_INITIAL`, `STATE_INITIALIZED`, `STATE_STARTED`, and `STATE_STOPED`) are mapped into one (`STATE_PRE_INIT`).

For entities that are part of candidate renamings REPENT performs data flow analysis. REPENT first builds a symbol table considering the entities scope, signature, and line number. Then, it identifies modifications and uses within and across files by resolving the imports of the files.

4.2 Candidate Renaming Filtering

The set of mapped entities computed in the previous step may contain false positives. For example, if a code fragment is moved from the top to the bottom of its file then the context *diff* may not trace it—produces incorrect results and several entity mappings may be created. The mapped entities cannot however be discarded outright if the old entity also exists in the new files. For example, a developer can move the body of a method into a new one (with a new method name); replacing the body of the old method with a call to the new method. Thus, REPENT must first assign a score to each entity mapping, then check if the entity in the old file also exists in the new one, and if so, compute a score for this new pair, and based on this latter score (compared to the other scores), keep or prune the other entity mappings. This latter step is referred in the following as "cross validation consistency check".

[6] This mapping is not desired and that the candidate renaming will be later filtered out due to the low similarity between the two entity declarations.

Fig. 6 reports details on the REPENT filtering strategy. Two cases may occur: (i) both entities in the candidate renaming have def-uses or (ii) at least one of the entities has no def-use. The second case is particularly common for getters and setters as they are often automatically generated and not necessarily used in the program.

For entities with def-uses, REPENT calculates the score of two entities involved in a candidate renaming as the textual similarity between the lines where the entity def-uses have been detected. For the entities without def-uses, the score is the textual similarity between the two entity declarations.

Finally, once scores are available, cross validation consistency check is performed. The following subsections provide details on how REPENT computes such scores and filters out likely false renamings.

4.2.1 Computing the Score between Entities with Def-uses

Let us assume that there are n statements in the old file where a given entity, say E_l, is either defined or used (or both). Also, let us assume that E_l has a candidate mapping with the entity E_k, and that in the new file there are m statements where the entity E_k is defined or used. To assign a score to the matching (E_l, E_k) REPENT creates an $n \times m$ *statement score matrix*. A matrix entry (i, j) contains the similarity score between statement s_i and s_j respectively of the old and new release. Before computing the score, REPENT removes the name of the entities from both statements (*i.e.*, s_i and s_j), to remove any bias introduced by similar entity names. The assigned score is based on the Normalized Levenshtein edit Distance (NLD) [37], defined as:

$$NLD(s_i, s_j) = \frac{LD(s_i, s_j)}{length(s_i) + length(s_j)} \tag{1}$$

where $LD(s_i, s_j)$ is the Levenshtein edit Distance between s_i and s_j. The score assigned to the statement pair (s_i, s_j), cell (i, j), of the *statement score matrix* is computed as $1 - NLD(s_i, s_j)$.

REPENT uses heuristics to prune low scores and thus reduce false positives. If the similarity is lower than a given threshold, named *Statement Similarity Threshold (SST)* (see Appendix B.1), the selection is filtered out by setting the (i, j) matrix entry to zero. REPENT uses different threshold values depending on the kind of entity. The reason behind this choice is the different nature of the definitions and uses of different entities.

As shown in Fig. 6, REPENT applies the Hungarian algorithm on the statement score matrix to identify the best possible matching between statements. The Hungarian algorithm [31] is an optimization algorithm used to solve the assignment problem, *e.g.*, assigning tasks to people, which given a score/cost matrix, will find the best assignment, *i.e.*, maximizing/minimizing the score/cost between matrix lines and columns.

The result of the Hungarian algorithm is used to assign a score to the pair of entities (E_l, E_k) as follows. If the number of mapped statements with scores higher than zero is higher than a predefined threshold—named *Number of Matched Statements Threshold (NST)*—then the score between the two entities is the sum of the similarities between the mapped def-uses; otherwise, the score is set to zero.

Consider the example in Fig. 5, which describes a many-to-one entity mapping where entities have def-uses. In this example, REPENT computes the scores for the following four possibilities:

```
STATE_INITIAL → STATE_PRE_INIT
STATE_INITIALIZED → STATE_PRE_INIT
STATE_STARTED → STATE_PRE_INIT
STATE_STOPED → STATE_PRE_INIT
```

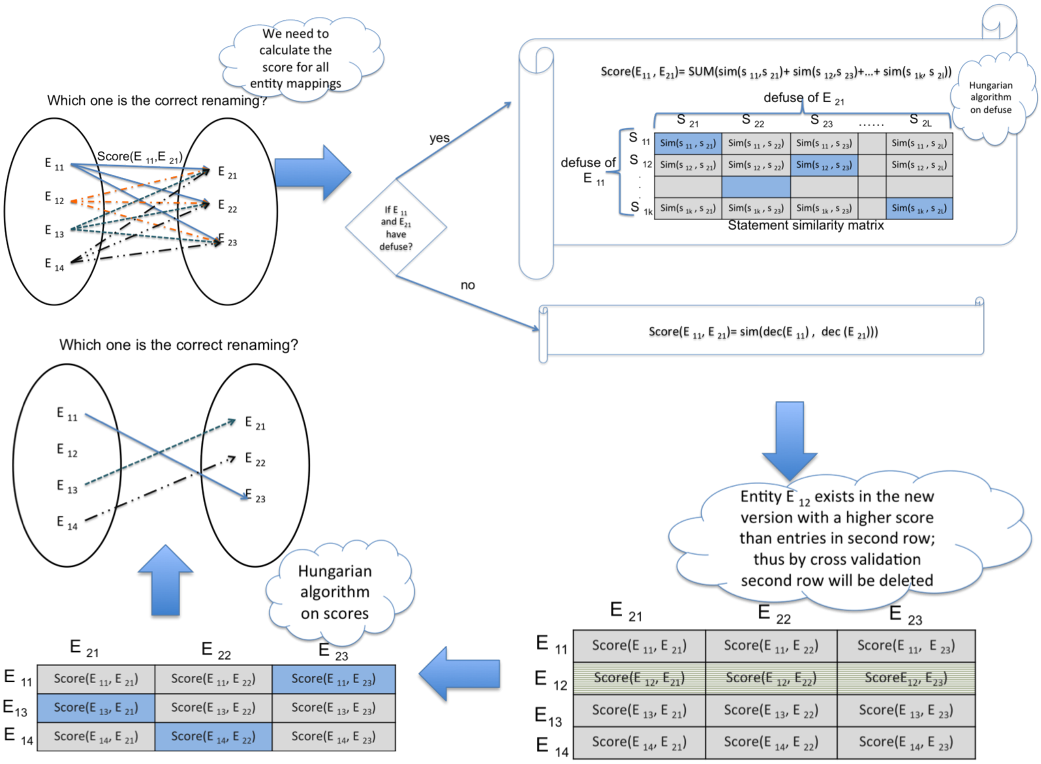
Fig. 7 illustrates the computation of the score for the first possibility, *i.e.*, `STATE_INITIAL` → `STATE_PRE_INIT`. In particular, once identified the def-uses, the entity names are removed and the similarity is computed between all the possible pairs of def-uses for the old and new entities. Such similarities are then stored in the *statement score matrix*.

The application of the Hungarian algorithm on the statement score matrix for the candidate renaming `STATE_INITIAL` → `STATE_PRE_INIT` creates the following statement matches: `O1` → `N1`, `O2` → `N2`, and `O3` → `N4`, where `Oi` (`Ni`) is the i^{th} statement of the older (newer) entity. The similarity scores of the matched statements for fields should be more than 0.8 according to Table 15. The heuristic to prune false renamings for fields states that one must have a number of matched statements higher or equal to 40% (see Appendix B.1) of the longest list of def-use statements for a candidate matching to be considered (this is to say, 40% of the largest score matrix dimension).

In the example described above, the matrix is a three by four matrix, which means that at least two def-uses must be matched ($4 \times 40\% = 1.6 \approx 2$). Actually, three statements have been matched, thus the score for the two entities (`STATE_INITIAL` and `STATE_PRE_INIT`) is the sum of the scores mapped by the Hungarian algorithm: 3.00 $(= 1.00 + 1.00 + 1.00)$.

4.2.2 Computing the Score between Entities without Def-uses

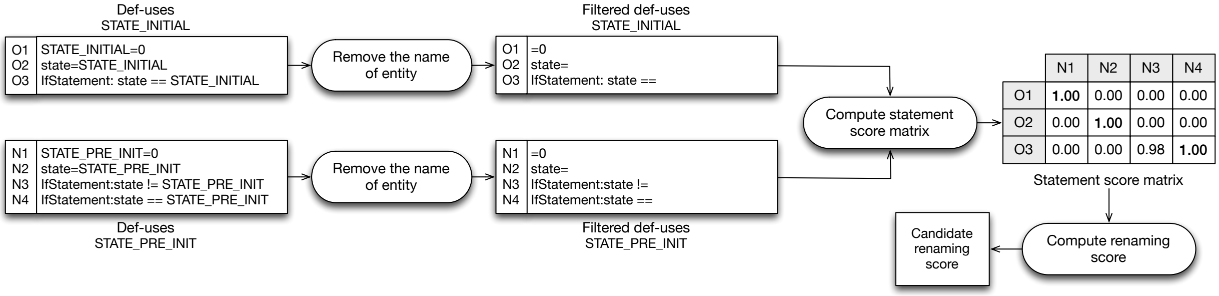
If at least one of the entity has no def-uses, the computation of the similarity is based on the textual


We need to calculate the score for all entity mappings
Which one is the correct renaming?
E_{11}
E_{12}
E_{13}
E_{14}
Score(E_{11},E_{21})
E_{21}
E_{22}
E_{23}
yes
If E_{11} and E_{21} have defuse?
no
Score(E_{11}, E_{21})= SUM(sim(s_{11},s_{21})+ sim(s_{12},s_{23})+...+ sim(s_{1k}, s_{2l}))
defuse of E_{21}
S_{21}
S_{22}
S_{23}
......
S_{2L}
defuse of E_{11}
S_{11}
S_{12}
S_{1k}
Sim(s_{11}, s_{21})
Sim(s_{11}, s_{22})
Sim(s_{11}, s_{23})
Sim(s_{11}, s_{2l})
Sim(s_{12}, s_{21})
Sim(s_{12}, s_{22})
Sim(s_{12}, s_{23})
Sim(s_{12}, s_{2l})
Sim(s_{1k}, s_{21})
Sim(s_{1k}, s_{23})
Sim(s_{1k}, s_{23})
Sim(s_{1k}, s_{2l})
Statement similarity matrix
Hungarian algorithm on defuse
Score(E_{11}, E_{21})= sim(dec(E_{11}) , dec (E_{21}))
Which one is the correct renaming?
E_{11}
E_{12}
E_{13}
E_{14}
E_{21}
E_{22}
E_{23}
Entity E_{12} exists in the new version with a higher score than entries in second row; thus by cross validation second row will be deleted
Hungarian algorithm on scores

	E_{21}	E_{22}	E_{23}
E_{11}	Score(E_{11}, E_{21})	Score(E_{11}, E_{22})	Score(E_{11}, E_{23})
E_{12}	Score(E_{12}, E_{21})	Score(E_{12}, E_{22})	ScoreE_{12}, E_{23})
E_{13}	Score(E_{13}, E_{21})	Score(E_{13}, E_{22})	Score(E_{13}, E_{23})
E_{14}	Score(E_{14}, E_{21})	Score(E_{14}, E_{22})	Score(E_{14}, E_{23})

	E_{21}	E_{22}	E_{23}
E_{11}	Score(E_{11}, E_{21})	Score(E_{11}, E_{22})	Score(E_{11}, E_{23})
E_{13}	Score(E_{13}, E_{21})	Score(E_{13}, E_{22})	Score(E_{13}, E_{23})
E_{14}	Score(E_{14}, E_{21})	Score(E_{14}, E_{22})	Score(E_{14}, E_{23})

Fig. 6. Candidate renaming filtering process.



	N1	N2	N3	N4
O1	**1.00**	0.00	0.00	0.00
O2	0.00	**1.00**	0.00	0.00
O3	0.00	0.00	0.98	**1.00**

Fig. 7. Computing the score of a candidate renaming in presence of entity def-uses.

similarity between the two entity declaration statements, ds_1—the declaration statement of the entity in the old file—and ds_2—the declaration statement of the entity in the new file. Also in this case, if the similarity $(1 - NLD(ds_1, ds_2))$ is lower than a predefined threshold—named *Declaration Similarity Threshold: (DST)*—we discard the candidate renaming by setting its similarity to the minimum, *i.e.*, to zero.

To better understand how the similarity is computed in this particular case, let us go back to the example of Fig. 4. In this example, only one candidate renaming has been identified, *i.e.*, `loadNative` → `terminate`. Because both methods have no def-uses, we simply compute the similarity between the declarations, *i.e.*, between `private void loadNative()` and `public native int terminate()`. The similarity between these two entities $(1 - NLD(ds_1, ds_2)$, where ds_1 is the declaration statement of `loadNative` and ds_2 is the declaration statement of `terminate`) is 0.63 which is lower than the fixed DST threshold (0.7); thus the score between `loadNative` and `terminate` is set to zero.

4.2.3 Computing the Score between Candidate Package Renamings

Using a package means importing that package or types declared in that package. Thus, a score based on def-uses is not suitable for package renamings as it does not allow one to distinguish between a case where types were moved from one package to another and a case where the package was actually renamed. In both cases, the use of the package will change in an identical manner; however, only the second situation is an actual package renaming. Thus, REPENT first computes the score of candidate package renamings using the version control system (to identify renamed, added, and deleted folders); then it filters out candidate package renamings that do not match a change in the folder structure of the version control system by setting their score to zero.

4.2.4 Filtering Out Likely False Renamings

As noted above, REPENT must also cross-check if entities exist in both files before promoting a candidate renaming to a real renaming. In essence, two further steps are needed: cross validation and entity score evaluation.

Cross validation consistency check: This step is necessary as there may be cases in which the line mapping is not precise. These cases often occur when source code fragments are moved within the same file (as in Fig. 4). Since line mapping relies on the Unix *diff*, and since *diff* relies of the context, *i.e.*, surrounding statements, such moving of code may result in wrong line mapping.

To cope with this imprecision, REPENT cross-checks each entity of a candidate renaming—*i.e.*, in the old file REPENT looks for an entity with the same name as the new entity and in the new version REPENT looks for an entity with the same name as the old entity—and computes the score between the two entities, using the algorithms described in Sections 4.2.1 and 4.2.2. If this score is greater than the score of the candidate renaming, then the renaming is discarded, *i.e.*, its score is set to zero. In the example in Fig. 4 the cross-checking identifies that `loadNative` → `terminate` is not an actual renaming, since the method `loadNative` is still present in the new file. Instead, in the example shown in Fig. 5, the cross-check fails since neither of the entities in the older version of the file appear in the new file and the entity of the new file does not appear in the old file.

Entity score evaluation: For each candidate renaming the scores of all possibilities are stored in $p \times q$ *entity score matrix*, where p and q are the number of old and new entities, respectively. Once such a matrix is produced for a candidate renaming, we apply the Hungarian algorithm to collect the assignments between the entities. Clearly, if a row (or a column) of the score matrix contains all zero values, the related entity is not renamed.

Fig. 8 shows the computation of the entity score matrix for the example in Fig. 5. This is a 4×1 entity score matrix where only one entry has a score greater than zero. Thus, when applying the Hungarian algorithm we obtain that the best score is 3.00, *i.e.*, the score between `STATE_INITIAL` and `STATE_PRE_INIT`. Therefore, REPENT concludes that `STATE_INITIAL` has been renamed to `STATE_PRE_INIT`.

4.3 Renaming Classification

The classification process of REPENT is summarized in Fig. 9. It entails a sequence of phases: (i) identifier splitting (Section 4.3.1), (ii) mapping of identifier terms (Section 4.3.2), and (iii) combining part of speech and semantic analyses (Section 4.3.3). Each phase is detailed in the following.

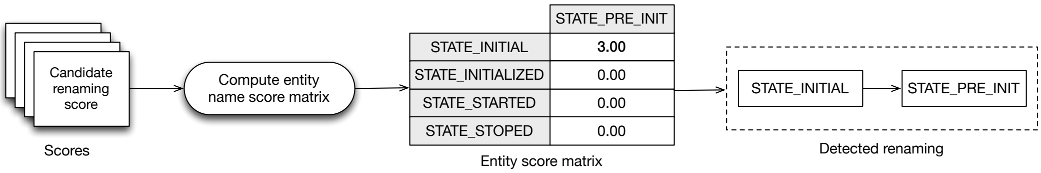
It is worth noting that the renaming classifier heavily relies on tools—ontological databases such as WordNet [43] and natural language parsers such as the Stanford Part-of-Speech Analyzer [53]—explicitly conceived to process natural language documents rather than source code. As pointed out by Hindle *et al.* [26], such tools can be far from optimal when applied to source code. However, at the moment they represent, to the best of our knowledge, the most suitable technology for our purposes. In future, REPENT could be further improved by replacing or combining WordNet with a domain-specific ontology. For example, in their recent work Yang and Tan [56] propose an approach to mine semantically related words in a project or multiple projects from the same domain. Similar work has been done by Howard *et al.* [27] where the authors mine semantically similar words across projects from multiple domains. However, in the current status we could not apply the aforementioned approaches, because they would have required us to manually validate all the mined semantic relations, which would have required a deep domain knowledge for the projects considered in our study (which we do not have).

4.3.1 Identifier Splitting

This step aims at splitting both the old and new names into their composing terms. REPENT uses a Camel Case splitting algorithm. The output of this phase is the lists of terms composing the old name *i.e.*, $t_{1,1}, t_{1,2}, \ldots, t_{1,n_1}$ and the new name *i.e.*, $t_{2,1}, t_{2,2}, \ldots, t_{2,n_2}$, where n_1 and n_2 are the number of terms composing the old and new names respectively. For example, the identifier `getChildCount` is split into `get`, `child`, and `count`. More sophisticated identifier splitting approaches such as *Samurai* [18], *TIDIER* [24], [38], *Normalize* [32], or *LINSEN* [13] can be plugged in. However, the current implementation of REPENT favors speed over accuracy; a Camel Case splitter is much faster than, for example, TIDIER [24], [38]. Moreover, previous studies found that for Java, the identifier splitting/expansion accuracy does not vary substantially between Camel Case and more sophisticated approaches [38].

4.3.2 Mapping Terms

The second phase aims at mapping the n_1 terms $t_{1,1}, t_{1,2}, \ldots, t_{1,n_1}$ composing the old name onto the



	STATE_PRE_INIT
STATE_INITIAL	**3.00**
STATE_INITIALIZED	0.00
STATE_STARTED	0.00
STATE_STOPED	0.00

Fig. 8. REPENT: Filtering false positives renamings.

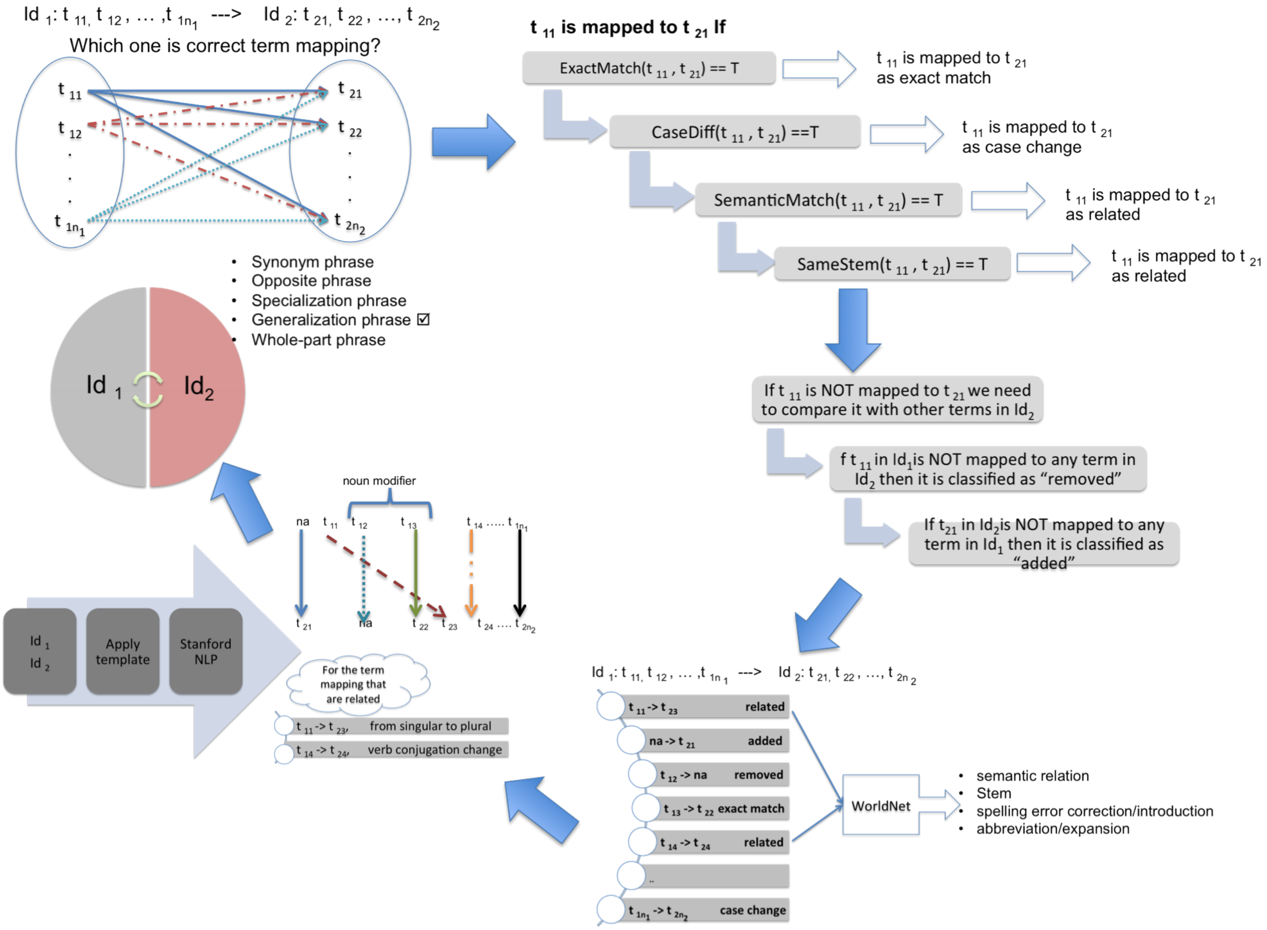

Id 1: t 11, t 12 , … ,t 1n1 ---> Id 2: t 21, t 22 , …, t 2n2
Which one is correct term mapping?
t 11
t 12
t 1n1
t 21
t 22
t 2n2
t 11 is mapped to t 21 If
ExactMatch(t 11 , t 21) == T
t 11 is mapped to t 21 as exact match
CaseDiff(t 11 , t 21) ==T
t 11 is mapped to t 21 as case change
SemanticMatch(t 11 , t 21) == T
t 11 is mapped to t 21 as related
SameStem(t 11 , t 21) == T
t 11 is mapped to t 21 as related
If t 11 is NOT mapped to t 21 we need to compare it with other terms in Id2
f t 11 in Id1is NOT mapped to any term in Id2 then it is classified as "removed"
If t21 in Id2is NOT mapped to any term in Id1 then it is classified as "added"
Synonym phrase
Opposite phrase
Specialization phrase
Generalization phrase ☑
Whole-part phrase
Id 1
Id2
noun modifier
na
t 11
t 12
t 13
t 14 ….. t 1n1
t 21
na
t 22
t 23
t 24 …. t 2n2
Id 1
Id 2
Apply template
Stanford NLP
For the term mapping that are related
t 11 -> t 23, from singular to plural
t 14 -> t 24, verb conjugation change
Id 1: t 11, t 12 , … ,t 1n 1 ---> Id 2: t 21, t 22 , …, t 2n 2
t 11 -> t 23 related
na -> t 21 added
t 12 -> na removed
t 13 -> t 22 exact match
t 14 -> t 24 related
..
t 1n1 -> t 2n2 case change
WorldNet
semantic relation
Stem
spelling error correction/introduction
abbreviation/expansion

Fig. 9. REPENT: Renaming classification process.

n_2 terms $t_{2,1}, t_{2,2}, \ldots, t_{2,n_2}$ of the new name. The mapping phase is, in turn, divided into two main steps depicted in Fig. 10. In the reported example REPENT must map the name `getStorage` onto `getMemoryBlock`.

First, REPENT discovers changed and unchanged terms plus added and deleted terms. To this aim, each term composing the old and new names is thought of as a source code line. In our example, the terms `get`, and `Storage` compose the lines of the first (old) file, while `get`, `Memory`, and `Block` represent the lines of the second (new) file. Then, a *diff* algorithm identifies churns of unchanged, added, removed, and changed terms (lines) between the two versions of a name (file), using an algorithm that solves the longest common subsequence (LCS) problem [14]. In the example in Fig. 10, after applying such an algorithm, the renaming of `getStorage` to `getMemoryBlock` is considered as the removal of the term `Storage` and the adding of the terms `Memory`, and `Block`. The term `get` is identified as unchanged.

In the second step, REPENT performs a fine-grained analysis of changed terms (*i.e.*, the term `Storage` from the old name and the terms `Memory`, and `Block` from the new name in the example shown in Fig. 10). Such an analysis is based on Algorithm 1 that builds a term-by-term mapping and classifies it. A term $t_{1,i}$ of the old name is mapped onto a term $t_{2,j}$ of new name according to multiple criteria, encoded in the function $matching(t_1, t_2, matchType)$. Given two terms and a matching criterion, this function returns *true* if the terms match according to the matching

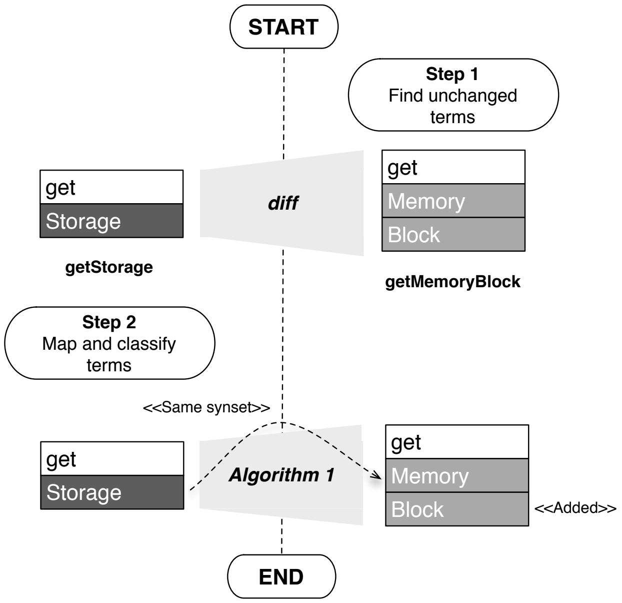


Fig. 10. REPENT: Term mapping and classification process.

criterion, *false* otherwise. Specifically, the matching is performed using, sequentially, the following criteria:

1) **Exact match**: if the two terms exactly match, *e.g.*, `get` and `get`.
2) **Case difference**: if the two terms only differ by the alphabetic letter case, *e.g.*, `Book` and `book`. If this does not happen, all terms are converted into lower case letter, and the subsequent criterion are matched.
3) **Semantic match**: if the two terms have any semantic relation according to the upper ontology WordNet. Words in WordNet are organized based on their relations. Synonyms are grouped into unordered sets, called synsets, which in turn are related using semantic and lexical relations. In the example reported in Fig. 10 the terms `Storage` and `Memory` belong to the same synset. The semantic relations considered by REPENT are synonym, hyponym, hypernym, antonym, meronym, and holonym. REPENT first identifies semantic relation between two mapped terms and if there is no such semantic relation it looks for a semantic relation between words in the synsets of the two mapped terms. This process repeats for up to three synsets of each word in the synsets of two mapped terms. For example, when looking if an antonym relation exists between two terms, REPENT first checks if there is an antonym relation. However, if this is not the case, REPENT further analyzes their respective synsets for antonym relation between two words, each belonging to the synset of the original terms. If an antonym relation is found, it will be considered as a relation of level 1. In the opposite case, *i.e.*, no relation is found, REPENT further looks for antonym relation between the words of the synsets of the synsets, *i.e.*, by doing a transitive closure up to level 3.
4) **Is stem**: if the two terms have the same stem according to the Porter [45] stemming. This rule is applied only if the *semantic match* rule fails. Indeed, if both terms are defined in WordNet, *e.g.*, `synchronization` and `synchronizing`, then they will be related according to the semantic match. If any of the two terms is not defined in WordNet, as it is the case of `invoc` from the identifier renaming `invocationType` → `invocType`, then the rule **is stem** is applied.

Algorithm 1 builds a mapping of terms of the old name onto any (not yet mapped) term of the new name, repeatedly traversing the terms of the new name, moving from the position of the term of the old name and using the above matching criteria, in the order in which they are mentioned.

After the term mapping has been performed, REPENT identifies mapped terms on which the renaming classification will focus, *i.e.*, all mapped terms that are not trivially mapped according to the **exact match** criterion. For example, in the identifier renaming `getStorage` → `getMemoryBlock`, both identifiers contain `get` that is an exact match and thus is removed from further consideration.

After terms of the old name have been mapped onto terms of the new name, REPENT classifies the renamings at term level, as:

1) **Removed:** terms of the old name not mapped onto any term of the new name are classified as removed, as it is the case for the term `statement` from the identifier renaming `statementLength` → `length`.
2) **Added:** terms of the new name not mapped onto any term of old name are classified as added. In the example reported in Fig. 10 the terms `Block` is identified as an added term.
3) **Matched:** terms of the old name mapped onto terms of the new name according to Algorithm 1 with an exact match, *e.g.*, the term `get` in the example reported in Fig. 10.
4) **Change case:** terms of the old name mapped onto terms of the new name according to Algorithm 1 with a case difference match, as this is the case for the term `jar` from the renaming `pJARFile` → `jarFile`.
5) **Related:** terms of the old name mapped onto terms of the new name according to Algorithm 1 with a **semantic match** or a **is stem** match. In the example reported in Fig. 10 the terms `Storage` and `Memory` are classified as related since there

```
foreach $matchType$ in ($exact$, $case\_difference$, $semantic$, $is\_stem$) do
    for $x \leftarrow 1$ to $n_1$ do
        if not $mapped_1[x]$ then
            $y1 \leftarrow x$, $y2 \leftarrow x$ ;
            while $y1 > 0$ or $y2 \leq n_2$ do
                foreach $y$ in ($y1$, $y2$) do
                    if matching($t_{1,x}$,$t_{2,y}$, $matchType$) and not $mapped_2[y]$ and $y > 0$ and $y \leq n_2$ then
                        $mapped_1[x] \leftarrow y$ ;
                        $mapped_2[y] \leftarrow x$ ;
                    end
                end
                $y1 - -$, $y2 + +$ ;
            end
        end
    end
end
```

Algorithm 1: Algorithm for mapping and classifying the n_1 terms of the old name onto the n_2 terms composing the new name.

exists a semantic relation (synonym) between them.

REPENT uses the mapped terms to classify the renaming in dimension *forms of renaming* as follows:

Simple: when only one term is added, removed, or changed.

Complex: when more than one terms are added, removed, or changed.

Formatting only: the following two conditions hold: (i) all term mappings are matched and/or change case and (ii) the two identifiers are the same when underscore and camel case are ignored.

Term reorder: the following two conditions hold: (i) at least two terms of the old identifier are matched to two terms in the new identifier while possibly changing case and (ii) the two identifiers are not the same when underscore and camel case are ignored.

REPENT refines **related** matches via WordNet to find semantic relations between terms, *i.e.*, synonymy, hyponymy, hypernymy, antonymy, meronymy, or holonymy and can thus classify the renaming in dimension *semantic change* as *synonymy*, *specialization*, *generalization*, *opposite*, or *whole-part* when the corresponding relation exists.

If no semantic relation is found REPENT checks whether there is a *spelling error* correction/introduction. REPENT assumes there is a spelling error if the following three conditions hold: (i) one of the two terms does not exist in WordNet but the other does, (ii) there is only small (string) difference between the two (Levenshtein distance is 2 or smaller—see Appendix B.2 for a discussion on the threshold value), and (iii) one term is not included in the other (to avoid misclassifying renamings such as `frame` → `jframe`).

Finally, if the previous checks fail, REPENT checks if there is an *abbreviation/expansion*. It assumes abbreviation/expansion if the following two conditions hold: (i) one of the two terms does not exist in WordNet but the other does and (ii) all characters of one are contained in the other.

4.3.3 Combining Part of Speech and Semantic Analyses

In natural language, a word carries a specific meaning. Words are often grouped into phrases which in turn can be combined to form sentences. The meaning carried by a phrase can narrow, generalize, or change the meaning of an individual term within the phrase. By analogy with natural language, to grasp the meaning of an identifier, one cannot rely only on the terms constituting the identifier in isolation. For example, the term `visible` (from the identifier `JavadocNotVisibleReference`) and the term `hidden` (from the identifier `JavadocHiddenReference`) have opposite meaning, whereas the identifiers have the same meaning.

Thus, after terms are mapped between the old and new name, REPENT explores the relations between the terms within the same identifier to classify identifier renamings. REPENT builds a synthetic sentence out of the identifier, then it performs a part of speech analysis.

As identifiers do not always follow well-formed grammatical structure, before applying part of speech analysis using natural language tools we apply a sentence template. Different templates have been proposed in the literature by Abebe *et al.* [3] and Binkley *et al.* [7]. For all kinds of entities, except methods, REPENT applies the List Item Template by Binkley *et al.* [7]. Indeed, they provided evidence that this template outperforms the other three templates they evaluated. For the identifier `inclusionPatterns` the template produces *inclusion patterns*. However, if the first term is a verb, as it is suggested according to Java standard for method names, REPENT uses a different template, *i.e.*, the verb template: "Try to <identifier terms>". A template is just an aid provided to the part of speech tagger to guide its analysis; thus for the method name `markAsDefinitelyUnknown`, REPENT applies the verb template on the term sequence, *i.e.*, *Try to mark as definitely unknown*.

REPENT part of speech analysis uses the Stanford Part-of-Speech Analyzer[7]. The Stanford NLP classifies

[7] We will refer to it as the Stanford NLP.

terms using the Penn Treebank Tagset [42], thus not only distinguishing between nouns, verbs, adjectives, and adverbs, but also distinguishing between the different forms. From this step beyond, we use the part of speech of each term—*i.e.*, whether it is a noun, being it singular or plural, an adjective, an adverb, etc.—and the relations between terms. More precisely, we are interested in the following relations: negation modifier (*i.e.*, the relation between a negation word and the word it modifies, as in the identifier `ignoreNotFoundField`), adjectival modifier (*i.e.*, a modifier relation between an adjective and a noun, meaning that the adjective specifies the noun, as in the identifier `binaryField`), and noun compound modifier (*i.e.*, a modifier relation between two nouns, meaning that one noun specifies the other, as in the identifier `methodSignature`).

Given a renaming pair, old and new names, REPENT processes the two part of speech analyses and uses heuristics to assign a semantic label to the renaming. The heuristics work as follows:

Synonym phrase: when the following two conditions hold: (i) there exists a term mapping where the two terms hold an antonym relation and (ii) one of the two terms is involved in a negation modifier relation.

Opposite phrase: when one of the following two conditions holds: (i) a negation modifier relation is added/removed while the modified term exists in both identifiers or (ii) a term renaming towards a synonym is accompanied with an addition/removal of a negation modifier relation.

Specialization phrase: when the following two conditions hold: (i) a term is added and (ii) it participates in a modifier relation, either adjectival or noun, with an already existing term.

Generalization phrase: when the following two conditions hold: (i) a term is removed and (ii) a modifier relation between the removed term and a term existing in both identifiers is also removed.

Whole-part phrase: when more than one term mapping pair holds a whole-part relation.

Add meaning: when the following two conditions hold: (i) there exists a term mapping that is classified as term added, and (ii) the added term is not the modifier of another term in the new identifier.

Remove meaning: when the following two conditions hold: (i) there exists a term mapping that is classified as term removed, and (ii) the removed term is not the modifier of a term in the old identifier.

Unrelated: when a term mapping does not fall into any of the levels of *semantic change*.

Part of speech change: when the part of speech of two mapped terms are different.

TABLE 2
Characteristics of the analyzed programs.

Program	Analyzed period	KLOCs (range)	Files (total)	File revisions	Committers
ArgoUML	1998-2012	1-20	300	68,400	42
dnsjava	1998–2011	9-35	365	1,415	2
Eclipse-JDT	2001–2006	2,089–6,949	5,758	54,571	50
JBoss	1999–2011	2,000-1,200	40,003	25,028	422
Tomcat	1999–2006	5–315	12,205	46,498	79

In the next two sections we discuss in detail two studies. The goal of the first study is to evaluate the accuracy of the renamings detected by REPENT (Section 5), while the second study is an exploratory study investigating the classification of the detected renamings with respect to the proposed taxonomy (Section 6).

5 Evaluating REPENT Detection Accuracy

This section reports the results of an empirical study conducted to evaluate the accuracy and completeness of REPENT renaming detection. The *goal* of this study is to analyze the detection accuracy of REPENT with the *purpose* of investigating to what extent undocumented renamings can be identified. The *perspective* of the study is that of researchers, who are interested in investigating how REPENT is suitable to identify renamings. The evaluation has been carried out in the *context* of the source code history of five Java open-source programs, namely ArgoUML, dnsjava, Eclipse-JDT, JBoss, and Tomcat. ArgoUML[8] is a UML modeling tool. dnsjava[9] is a Java Domain Name System (DNS) implementation. Eclipse-JDT is a set of plug-ins that adds the capabilities of a full-featured Java IDE to the Eclipse[10] platform. JBoss AS[11], in the following simply referred to as JBoss, is a Java application server. Tomcat[12] is an implementation of a servlet container and Java Server Page (JSP) engine.

Table 2 reports the main characteristics of the analyzed programs: the analyzed time periods, size ranges in KLOCs, numbers of Java files, numbers of analyzed revisions, and numbers of committers. ArgoUML and Eclipse are versioned under CVS, while all other programs are versioned under SVN. The analyzed programs cover different domains while sizes, numbers of files as well as numbers of analyzed versions are spread fairly evenly in a broad range of small, medium and large programs.

Table 3 reports the total user time needed by REPENT to detect renaming throughout the evolution history

[8] http://argouml.tigris.org
[9] http://www.xbill.org/dnsjava
[10] http://www.eclipse.org
[11] http://www.jboss.org/jbossas
[12] http://tomcat.apache.org

TABLE 3
REPENT time of detection.

Program	Total time (min)	Average time for 2 revisions of a file (sec)
ArgoUML	113	0.23
dnsjava	11	0.14
Eclipse-JDT	561	0.33
JBoss	780	0.47
Tomcat	117	0.21

of the five projects (hardware: 16 Intel(R) Xeon(R) CPUs @2.40GHz). This time is purely indicative, as the proof of concept detector has not been optimized to reduce the required time. A single file revision requires a very short time (third column of Table 3), which permits the on-line use of the detector. By contrast, a complete analysis of a system evolution history should be performed offline since it may require hours.

The following subsections detail the procedure and results of the accuracy evaluation of REPENT, in particular we discuss precision and recall.

5.1 Research Questions and Study Procedure

To evaluate the accuracy of REPENT, we address the following two research questions:

RQ-DP: *How accurate is the set of renamings detected by* REPENT? This research question aims at estimating the accuracy of the detection approach, measured in terms of *precision*. Since Section 6 presents an empirical study on how developers rename identifiers, precision indicates the accuracy of the renamings used in such a study.

RQ-DR: *How complete is the set of renamings detected by* REPENT? This research question aims at estimating the completeness, measured in terms of *recall*, of REPENT with respect to the set of renamings performed by the developers of the analyzed programs. Recall gives an estimate of the representativeness of the analyzed renamings reported in Section 6.

The following subsections detail how we evaluate the accuracy of REPENT.

5.1.1 Evaluating the Precision of the Detection Approach: Manual Validation

Precision is computed by manually validating a sample of renamings from the analyzed programs.

We reuse the oracle built from our previous work, *i.e.*, manually validated renamings for Tomcat, to calibrate thresholds (see Appendix B.1). We then evaluate the approach on all programs by validating a statistically representative sample for each. Sampling separately for each program allows us to evaluate the detection also on programs with low number of renamings (*e.g.*, dnsjava), which otherwise would have less chances to be selected (*e.g.*, with respect to JBoss) if the total population was considered.

To estimate the size of the representative samples we choose a confidence level of 95% and a confidence interval of $\pm 5\%$ [49]. Thus, we can be 95% sure that the precision of the approach on the detected renamings for each program will be the precision estimated for the sample $\pm 5\%$.

Once the sample sizes have been determined, we use a stratified random sampling to select the renamings to be validated. This means that for each program we first group renamings based on the kind of entity being renamed, *i.e.*, the first dimension of our taxonomy (*e.g.*, type, method). Then, we estimate the proportion of each group with respect to the total population of detected renamings for that particular program and we use the same proportion for the sample. For example, if type renamings are 5% of the total population of detected renamings in ArgoUML, 5% of the sample must be type renamings. Finally, we randomly select the sample for each group. The sample size and the number of detected renamings are reported in Appendix C.1. The advantage of using stratified random sampling is in ensuring that all groups are represented [8]. If random sampling is used instead, the chances that a package renaming is selected for validation for Eclipse-JDT for example are almost zero (4 over 12,557).

The manual validation was conducted as follows. For each chosen detected renaming, two of the authors of this paper independently inspected the source code of both versions of the file between which the entity was renamed. All available details (comments, uses, neighboring entities) contributed to the decision-making. Then, each author marked the renaming as true positive (TP) or false positive (FP). In all cases in which the two authors provided a different classification for the renaming, the inconsistency was discussed and solved. When needed, a third author also reviewed such candidate renamings in which case the classification was obtained by a majority vote. Whenever the lack of knowledge prevented us from taking a decision, the renaming was removed and replaced by a new one; the process was iterated up to the required sample size.

Finally, the precision is computed as the fraction of detected renamings in the validated sample that the authors of this paper classified as TP. In other words, given the subset of detected renamings sampled for validation, TPS the set of those classified as true positives, and FPS the set of those classified as false positives, the precision Pr is given by:

$$Pr = \frac{|TPS|}{|TPS| + |FPS|}$$

5.1.2 Evaluating the Recall of the Detection Approach: Comparison with Documented Renamings

To evaluate the recall, ideally one should have the knowledge of all actual renamings that occurred in a program. Unfortunately, such information is not available for open-source programs. However, there are (relatively few) cases in which developers documented renamings in the versioning system commit notes. Hence, we estimate the recall as the proportion of such documented renamings also detected by REPENT.

To identify documented renamings, we filter the commit logs and consider only the commits whose note contain the keyword "renam". Then, we complement the automatic filtering with a manual analysis of the identified commit notes, with the aim of pruning out false positives. Typical cases of false positives are the commits in Eclipse-JDT related to changes to the refactoring feature, which includes a renaming feature. Other false positives are related to renaming of files not containing source code, *e.g.*, images or documentation files. Then, we analyze the source code of the files involved in the documented renaming to locate renamed entities, and hence verify whether such renamings are detected by our approach. Hence, given DCR, the set of documented renamings identified as described above and DR the set of detected renamings, the recall Rc is the proportion of documented renamings that are also detected by REPENT:

$$Rc = \frac{|DR \cap DCR|}{|DCR|}$$

5.2 Analysis of the Results

This section analyzes the results achieved aiming at answering our research questions **RQ-DP** and **RQ-DR**.

5.2.1 RQ-DP: How accurate is the set of renamings detected by REPENT?

Table 4 reports, for the analyzed programs, the precision of REPENT computed for each kind of entity as well as for the overall sample of renamings.

Overall, we observed an average precision of about 88%, as expected slightly lower than the one computed when calibrating the thresholds (about 92%, see Table 15). The number of detected package renamings is very low, thus at most 1 package renaming was sampled per program, which results in a 0% or 100% precision and explains the lowest precision reported in Table 4, *i.e.*, 67% for package renamings. REPENT has a somehow low precision—compared to the rest of the entity kinds—for parameters renamings (76%). The set used to calibrate the thresholds may not have a sufficiently large set of parameter renaming and thus thresholds may need to be recalibrated. The accuracy of REPENT may also be impacted by methods not being called in the program, or getters/setters automatically generated and again not used. In such cases REPENT relies on the fixed DST string matching threshold. Higher values may improve precision but again at the cost of recall.

TABLE 4
Estimated precision Pr for renaming detection of different entities (95% ± 5 confidence).

	ArgoUML	dnsjava	Eclipse-JDT	JBoss	Tomcat	Overall
Package	0%	-	100%	100%	-	67%
Type	100%	100%	100%	100%	100%	100%
Constructor	100%	100%	100%	100%	100%	100%
Field	100%	93%	95%	94%	73%	93%
Method	100%	98%	94%	94%	93%	94%
Getter/Setter	100%	83%	97%	96%	79%	90%
Parameter	90%	54%	92%	87%	67%	76%
Local variable	95%	86%	93%	84%	90%	92%
Overall	**97%**	**78%**	**94%**	**91%**	**80%**	**88%**

TABLE 5
Comparison with documented renamings.

Program	Files involved	Documented renamings
ArgoUML	77	4
dnsjava	113	229
Eclipse-JDT	140	52
JBoss	146	50
Tomcat	66	2

5.2.2 RQ-DR: How complete is the set of renamings detected by REPENT?

Table 5 reports the number of files involved in the commits whose log message suggest possible renamings. Documented renamings refer to the number of renamings that we found in the files committed with the log messages that were either documenting a renaming in a vague manner (*e.g., "renamed some stuff"*) or explicitly (*e.g., "rename Name.fromStringNoValidate(String) to Name.fromStringNoException(String)"*).

Table 6 reports—for each kind of entity—the detected proportion of documented renamings. We conclude that although sometimes renamings are documented, this is not a general rule. This result further motivates the use of REPENT as a renaming re-documentation tool. Table 6 also shows that documented renamings often pertain to types and, thus, to constructors.

For ArgoUML, the number of documented renamings is very low. We detect three out of four renamings. The renaming our approach fails to detect is a complex

TABLE 6
Detected documented renamings and recall Rc of different entities.

	ArgoUML		dnsjava		Eclipse-JDT		JBoss		Tomcat		Overall	
Package	-		-		100%	(1/1)	-		-		100%	(1/1)
Type	-		94%	(58/62)	18%	(4/22)	95%	(20/21)	-		78%	(82/105)
Constructor	-		100%	(134/134)	100%	(4/4)	95%	(18/19)	-		99%	(156/157)
Field	100%	(1/1)	100%	(4/4)	100%	(3/3)	100%	(1/1)	-		100%	(9/9)
Method	-		100%	(1/1)	100%	(7/7)	100%	(5/5)	-		100%	(13/13)
Getter/Setter	67%	(2/3)	-		-		100%	(2/2)	100%	(2/2)	86%	(6/7)
Parameter	-		100%	(28/28)	100%	(7/7)	-		-		100%	(35/35)
Local variable	-		-		88%	(7/8)	100%	(2/2)	-		90%	(9/10)
Overall	75%	(3/4)	98%	(225/229)	63%	(33/52)	96%	(48/50)	100%	(2/2)	92%	(311/337)

combination of renaming and refactoring activities, where the renamed getter method was abstract in the old version and became a concrete method in the new version. Also, the field associated with the getter was moved from the superclass to a subclass.

In Tomcat there are only two documented renamings and REPENT detects both of them.

For JBoss, REPENT only fails to identify two documented renamings, *i.e.*, one class and one constructor. Both entities are defined in the same file and the file was renamed as well. REPENT misses these renamings as the difference between the original and renamed files is greater than the 60% relative threshold for detecting renamed files.

For Eclipse-JDT, REPENT fails to identify one local variable and 18 class renamings. mainly because Eclipse-JDT used (for the analyzed period) CVS. Therefore, as explained in Section 4, we grouped commits using the heuristic of Zimmermann *et al.* [57] However, sometimes commits belonging to the same change occur in different days and developers do not always use consistent commit notes. As a consequence, REPENT fails to identify some file renamings.

Finally, dnsjava was (surprisingly) the program containing the highest number of documented renamings, despite being the smallest one. In this case, our approach detected 98% of the documented renamings, *i.e.*, it fails to detect only four class renamings. These classes have no def-uses. Although REPENT detects these class renamings as candidate renamings, it filters them as false positive, since their similarities are less than the declaration similarity threshold for type renamings (see Table 15).

Table 7 summarizes REPENT precision and recall. The first column reports the data set used for the evaluation; the second column corresponds to the size of the data set (*i.e.*, numbers of renamings); the last column reports the measures. Precision was evaluated over a representative sample from the detected renamings while recall was evaluated with respect to renamings documented by developers. The documented renaming set is extracted from the repository of the programs under analysis. Although its size is not as large as the sample used to evaluate the precision, it is an unbiased oracle as the entries are reported by the developers.

TABLE 7
REPENT precision and recall.

Data set	Size	Accuracy (measure)
Sample over detected renamings	1723	88% (Pr)
All documented renamings	337	92% (Rc)

> **REPENT detection accuracy:** Overall, we found that REPENT reaches high precision (88%) and recall (92%) thus being suitable for most of the foreseeable tasks.

6 REPENT IN ACTION: HOW RENAMINGS FOLLOW THE TAXONOMY

The *goal* of this study is to use REPENT to analyze renamings over the evolution history of software programs with the *purpose* of investigating to what extent such renamings fall into the dimensions defined in the taxonomy of Section 3. The *perspective* of the study is that of researchers who are interested in investigating how identifiers are renamed in the same *context* as the study reported in Section 5.

6.1 Research Questions and Study Procedure

This empirical study aims at automatically detecting and classifying renamings in the five programs described in Table 2.

Since we use REPENT to identify renamings, we analyze the classification accuracy of REPENT to evaluate to what extent the classification of renamings with respect to our taxonomy is affected by the performance of REPENT, thus answering the following research question:

RQ-CP: *How accurate is the set of renamings classified by* REPENT? This research question aims at providing an estimate of the accuracy of the classification, measured in terms of *precision*. Such an estimate indicates the accuracy of the results of this exploratory study, reported in Section 6.2.

While the focus of the previous research questions is to evaluate the reliability of REPENT as a tool to detect and classify identifier renamings, the following research questions are the core of this study, *i.e.*, they study the renaming phenomenon using the taxonomy defined in Section 3. For each dimension of the taxonomy, we investigate to what extent renamings of the five programs fall into the different levels of the dimension.

RQ1: *To what extent do renamings occur with respect to the different kinds of entities?* Specifically, we compute the number and proportion of renamings occurring for package, type, constructor, method/getter/setter, field, parameter, and local variable names to investigate which entities are more prone to be renamed.

RQ2: *What kinds of changes occur to terms composing identifiers when these are renamed?* In other words, we compute the number and proportion of simple, complex, formatting only, and term reordering renamings to investigate which forms are more frequent.

RQ3: *What kinds of semantic changes occur in identifiers when they are renamed?* In other words, we compute the number and proportion of renamings that preserve, change, narrow, broaden, add, and remove meaning to study how the renamings of the five programs are distributed over the different levels of semantic change.

RQ4: *What kinds of grammar changes occur in identifiers when they are renamed?* Specifically, we investigate to what extent the renamings imply changes to nouns (singular/plural), to verb conjugations, or other part of speech changes.

In order to find answers to our research questions, we investigate—from both a quantitative and qualitative point of view—how identifier renamings detected in the studied programs follow the taxonomy of Section 3.

6.1.1 Evaluating the Precision of the Classification Approach: Manual Validation

To evaluate the accuracy of the renaming classification we extract, for each level of each dimension of the taxonomy, a representative random sample ensuring a confidence interval of ±10% for a confidence level of 95%. This is different from the sampling in Section 5 where the sample is representative for each program stratified over the kinds of entities. Here, the confidence level and interval criteria are met for each level and each dimension of the taxonomy for the total population of classified renamings. For the *semantic change* dimension this means a representative number of expansions, a representative number of abbreviations, etc.

6.2 Analysis of the results

In this section we first discuss how accurately REPENT classifies renamings (Section 6.2.1), then, in Sections 6.2.2 to 6.2.5, we discuss how the renamings of the five programs that we studied follow the taxonomy defined in Section 3, *e.g.*, to what extent those renamings *preserve meaning* or consist of changes that are *formatting only*.

6.2.1 RQ-CP: How accurate is the set of renamings classified by REPENT?

We manually analyzed a sample of the classified renamings to evaluate in how many cases REPENT correctly or wrongly classified the changes in the identifiers. In addition, when REPENT fails to correctly classify a change we further investigate the reason. The sample size and the number of correctly classified renamings for each dimension of taxonomy are reported in Tables 18 to 20 in Appendix C.2.

With respect to the classification of *forms of renaming*, REPENT has an overall precision of 98% (see Table 18). The few misclassified cases are due to wrong term mapping.

For the classification of *semantic changes*, REPENT exhibits an accuracy of 80% (see Table 19). REPENT is very accurate in classifying renamings that add or remove meanings, 82% and 91% respectively. REPENT is also accurate in classifying renamings that preserve the meaning (overall precision of 93%). The lowest accuracy is achieved by REPENT when classifying renamings as *narrow* and *broaden meaning*, 62% and 69% respectively. Wrongly classified renamings in the category of *semantic changes* are due to wrong splitting, wrong term mapping, or wrong relations between terms. We also observed cases where REPENT misclassified a renaming because of the ontological database. For example, WordNet infers a hyponym relation between "is" and "get" and an antonym relation between "long" and "short". Those relations are not valid in the context of Java where in many cases "is" and "get" are used for accessors of boolean attributes and where "long" and "short" are primitive types. Approaches by Yang and Tan [56] and Howard *et al.* [27] can be used to improve the classification of *semantic changes*.

The classification accuracy of REPENT when classifying *grammar changes* is 74% (see Table 20). REPENT is accurate in classifying changes of nouns from singular to plural (and *vice versa*) and changes in verb conjugation, *i.e.*, the precision is 100% and 79% respectively.

TABLE 8
Renamed entities identified by REPENT.

	Package	Type	Field	Constructor	Method/Getter/Setter	Parameter	Local variable	Total
ArgoUML	0	18	2,156	16	391	690	712	3,983
dnsjava	0	67	58	159	219	448	144	1,095
Eclipse	4	180	1,942	139	3,205	3,218	3,845	12,533
JBoss	7	656	1,805	475	3,985	3,406	3,247	13,581
Tomcat	0	69	478	48	830	507	428	2,360
Total	**11**	**990**	**6,439**	**837**	**8,630**	**8,269**	**8,376**	**33,552**

Moreover, REPENT is very accurate where there is no grammar change (100%). By contrast, REPENT performances are fairly low (20% of precision) in classifying changes in other part of speech changes. Most of these cases are mainly due to the Stanford NLP not being accurate when parsing source code identifiers. In a very recent work, Gupta *et al.* [25] proposed an approach for part of speech tagging of source code identifiers and showed that the approach parses identifiers 10% to 20% more accurately. Unfortunately, at current date, the source code identifier tagger is not publicly available. However, it could be integrated in REPENT to improve its performances. Other reasons for misclassification were incorrect splitting or incorrect term mapping.

REPENT classification accuracy: REPENT almost perfectly classifies forms of renamings (98%) and classifies reasonably well *semantic changes* (with an accuracy of 80%). The lowest performance is on the classification of *grammar changes* (with an overall accuracy of 74%).

6.2.2 RQ1: To what extent do renamings occur with respect to the different kinds of entities?

Table 8 and Fig. 11 report the number and proportion, respectively, of renamings occurring for package, type, field, constructor, method/getter/setter, parameter, and local variable names[13].

The entities that are more prone to be renamed are methods, parameters, and local variables. Generally, such changes reflect the evolution of the programs. Indeed 89% of the surveyed developers confirm that they rename while changing functionality. As an example in JBoss, REPENT identified that the parameter `webserviceClientDeployer` has been renamed to `webservicesClient`. The renaming was performed to reflect a change in the functionality, as confirmed by the log message: *decouple WebserviceClientDeployer from JSR109ClientService.*

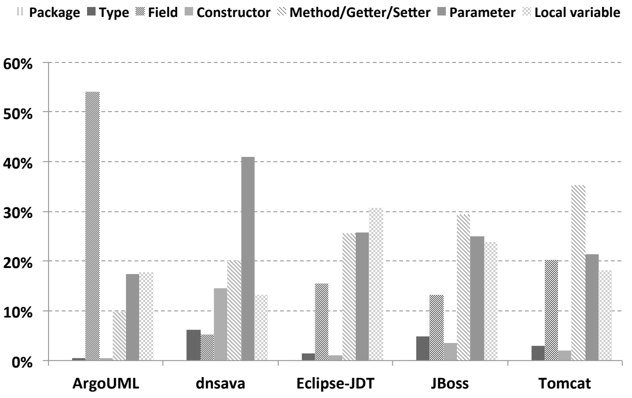


Fig. 11. Proportion of renamed entities identified by REPENT.

There is a large number of field renamings in ArgoUML. In this program, REPENT identified 2,156 field renamings, representing about 54% of the renamings. About 67% of the field renamings consisted solely of underscore removal in the beginning of the field, *i.e.*, renaming away from the Hungarian notation. Finally, 11% of the field renamings were performed due to the third party library that ArgoUML uses for logging purposes, *i.e.*, Log4J. The class `org.apache.log4j.Category` was deprecated and users of the library were supposed to use class `org.apache.log4j.Logger` instead. As a result, fields were renamed from `cat` to `LOG` or `logger`.

Finally, 448 out of 1,095 of the renamings in dnsjava were performed on method parameters due to massive renaming activities that heavily changed the API and hence the method parameters. Also, a considerable number (20%) of the parameter renamings consisted solely in removing a leading underscore. Conversely, in 52% of the parameter renamings (all being renamed in the same revision) a leading underscore was added. In addition, the majority of those parameters was renamed following the same pattern: names starting with the letter `r` were renamed to start with underscore. Examples of those include `rname` → `_name`, `rclass` → `_dclass`, and `rtype` → `_type`. All those renamings were performed as part of *"The big rewrite..."* in class `Record`, a generic resource record, or in one of its many subclasses. About 6% of the parameter renamings were performed on parameters of type `DataByteOutputStream` where the name changed from `dbs` to `out`.

[13] For the classification of renamings we only considered the TP renamings from the validated sample, thus the number of classified renamings is lower than the number of detected renamings .

TABLE 9
Forms of renamings identified by REPENT.

	Simple	Complex	Formatting only	Term reordering	Total
ArgoUML	1,787	493	1,702	1	3,983
dnsjava	894	85	116	0	1,095
Eclipse-JDT	7,910	4,456	132	35	12,533
JBoss	8,094	4,786	655	46	13,581
Tomcat	1,638	658	58	6	2,360
Total	**20,323**	**10,478**	**2,663**	**88**	**33,552**

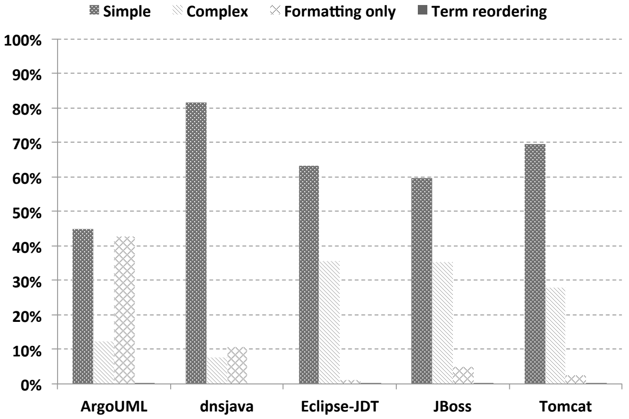


Fig. 12. Proportion of forms of renamings identified by REPENT.

RQ1 conclusion: Renamings occur mostly for method, parameter, and local variable names, with 26%, 25%, and 25% of the renamings respectively. Field renamings represent also a large proportion of the renamings—close to 20%. Finally, type renamings, which in Java imply constructor renamings, as well as package renamings, represent a small proportion of the renamings (less than 3% each for type and constructor renamings, less than 1% for package renamings).

6.2.3 RQ2: What kinds of changes occur to terms composing identifiers when these are renamed?

Table 9 reports the forms of identified renamings by REPENT, while Fig. 12 shows the proportion of the different forms. Most of the renamings were classified as *simple* renamings, where developers renamed a single term. In Eclipse-JDT, JBoss, and Tomcat, there is a considerable number of *complex* renamings while this form of renamings is not so frequent in the other two programs.

In ArgoUML, a substantial number of renamings is consist of *formatting only*—43% of the identified renamings. The analysis of this form of renamings indicates that in 77% (1,314 out of 1,702) of the cases, the renaming relates to the removal of leading underscores from identifiers, *i.e.*, towards Java naming conventions, which recommend not to start identifiers with underscore. However, in 2% (35 out of 1,702) of those renamings, a leading underscore was added to identifiers, hence, against Java naming conventions.

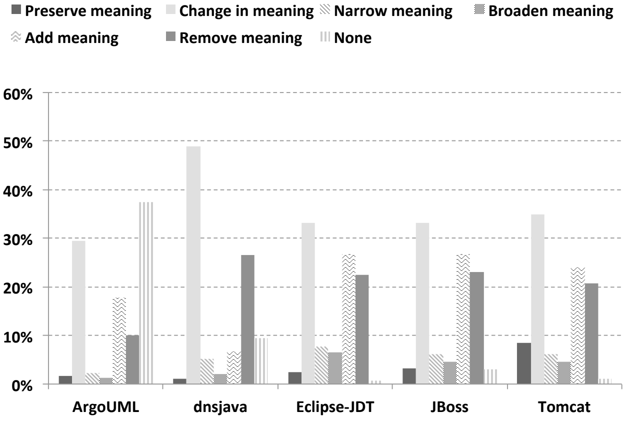


Fig. 13. Proportion of semantic changes identified by REPENT.

These results go along with the opinion of surveyed developers—when the name of an entity does not follow the language naming conventions, 34% would definitely rename while 46% would probably rename.

REPENT identified only a few renamings (88 of the classified renamings) that were performed to change the order of terms. One may expect that *term reordering* involves entities with limited scope, thus limiting the impact of the renaming. However, in the analyzed renamings only 15% of such re-orderings involved local variables. We conjecture that developers tend to reorder terms to improve the comprehensibility of the identifier and avoid misunderstanding. For example, in JBoss a developer changed a method parameter name from `serviceDest` to `destService`, to clarify that the parameter contains the address of the destination service.

RQ2 conclusion: The majority of renamings (61%) are simple renamings, *i.e.*, only one term of the identifier is renamed. In a considerable number of the renamings (31%) multiple terms are changed simultaneously. Less often (8%), renamings consist only of formatting changes. Even less often are those renamings where the terms of the identifiers were reordered.

6.2.4 RQ3: What kinds of semantic changes occur in identifiers when they are renamed?

Table 10 reports the number of semantic changes identified by REPENT while Fig. 13 shows the proportion of semantic changes.

Renamings that *preserve meaning* are quite unusual in the analyzed programs. Table 11 shows detailed results for this category of renamings. There is a number of renamings classified as *spelling error*. As expected, most renamings correct spelling errors while only a small number introduce spelling errors.

TABLE 10
Semantic changes identified by REPENT.

	Preserve meaning	Change in meaning	Narrow meaning	Broaden meaning	Add meaning	Remove meaning	None	Total
ArgoUML	77	1,311	97	59	789	441	1,661	**4,435**
dnsjava	12	576	62	24	79	312	112	**1,177**
Eclipse	413	5,522	1,297	1,104	4,481	3,750	122	**16,689**
JBoss	580	6,042	1,130	851	4,884	4,198	564	**18,249**
Tomcat	259	1,061	186	138	731	628	35	**3,038**
Total	**1,341**	**14,512**	**2,772**	**2,176**	**10,964**	**9,329**	**2,494**	**43,588**

TABLE 11
Preserve meaning renamings as classified by REPENT.

	Synonym	Synonym phrase	Spelling error correction/introduction	Expansion	Abbreviation	Total
ArgoUML	10	0	12	44	11	77
dnsjava	0	0	1	9	2	12
Eclipse	163	1	137	61	51	413
JBoss	195	2	180	84	119	580
Tomcat	185	0	34	11	29	259
Total	**553**	**3**	**364**	**209**	**212**	**1,341**

301 out of the 364 spelling error renamings are simple renamings indicating that spelling errors are corrected in isolation. Some renamings aim at correct spelling errors but even after multiple corrections, the identifiers still contain spelling errors (*e.g.*, `defferedSyntaxAllowedAsLitteral` → `deferedSyntaxAllowedAsLitteral` → `deferedSyntaxAllowedAsLiteral`) because only few available IDEs (*e.g.*, EMACS, ECLIPSE) provide support for spell-checking of identifiers.

We expected that renamings towards *expansions* would be performed for clarification purposes, *e.g.*, `getAlg` → `getAlgorithm`. 56% of such expansions concern local variables, which indicates that entities with limited scope are also important and developers take care of them. However, the overall number of renamings towards expansions is low (209): 49% of the surveyed developers would probably not undertake a renaming if the name of an entity contains an abbreviation or an acronym; 7% would definitely not rename; 30% were undecided; and only 13% would probably rename. As for abbreviations, we expected that *abbreviation* renamings would occur when identifiers are long and are composed of many terms. Yet, in more than 75% of such renamings, the old names are composed of only one or two terms. For example, the parameter `parameters` in JBoss was renamed to `params`, while the local variable `association` in ArgoUML was renamed to `assoc`.

REPENT identified 3 cases of *synonym phrase*. Two fields were renamed from `NOT_CLOSED` to `OPEN`; the third renaming is a false positive. We also manually found an example in Eclipse-JDT, where `javadocNotVisibleReference` was renamed to `javadocHiddenReference`. REPENT failed to correctly classify this renaming because the Stanford NLP wrongly assigns the negation relation (due to the term `Not`) to the term `javadoc` instead of assigning it to the term `Visible`.

Renamings that *change meaning* are the most frequent. In general, 33% of the renamings aim at changing the meaning of the identifiers. Such renamings are particularly frequent in dnsjava—48% of the semantic changes. As explained in Section 6.2.2, dnsjava underwent a massive renaming (*e.g.*, `rname` becomes `_name`). Those cases are classified as *change meaning* as REPENT fails to relate the meaning of the two terms due to the non-use of separator in the case of `rname`. Here, REPENT would benefit from a more sophisticated splitting technique that would split the identifier into two terms, *i.e.*, `r` and `name`. ArgoUML also underwent a massive renaming activity, due to the use of Log4J as explained in Section 6.2.2. While in the context of ArgoUML, REPENT classifies such renamings as *change in meaning*, we suspect that for the developers of the third-party library (Log4J) the terms `Category` and `Logger` have the same meaning, the former being a superclass of the latter. If this is indeed the case, a domain dictionary would improve the classification.

Table 12 shows the results of *change meaning* renamings at a fine-grained level—according to the proposed taxonomy. In general, there is no semantic relationship, *i.e.*, *unrelated* according to taxonomy, between the renamed terms in this category. As an example, in ArgoUML, REPENT identified that a parameter name was changed from `eventNames` to `propertyNames` to reflect the new semantics of the parameter. However, although quite rare, there are cases where the developers inverted the responsibility of an entity, *e.g.*, in JBoss REPENT identified that a method name was changed from `isInvisibleAnnotationPresent` to `getVisibleAnnotation` to reflect the new behavior of the method. REPENT identifies only two identifiers where the names changed from `body` to `node`, *i.e.*, renamings where the semantic change is a whole-part phrase. This type of renaming is less likely to occur.

Particularly interesting are renamings that involve identifiers that contain a negation. Such identifiers are usually renamed towards positive names; this is a particular example of *opposite phrase* renamings identified by REPENT. For example, in Eclipse-JDT the method `isNotPrimitiveType` was renamed to `isPrimitiveType` and the local variable `dontSetFigs` of ArgoUML was renamed to `setFigs`. From the analysis of the entities involved in such renamings, we

TABLE 12
Change in meaning renamings as classified by REPENT.

	Opposite	Opposite phrase	Whole-part	Whole-part phrase	Unrelated	Total
ArgoUML	0	2	0	0	1309	1,311
dnsjava	0	0	0	0	576	576
Eclipse	44	29	0	0	5449	5,522
JBoss	29	38	0	0	5975	6,042
Tomcat	16	7	2	0	1036	1,061
Total	**89**	**76**	**2**	**0**	**14,345**	**14,512**

observed that they are usually used with the negation operator. In such cases it is more difficult to interpret an expression containing such entities, especially if the expression contains a Boolean negation operator. However, among the surveyed developers only 30% would rename an entity if the name contains a negation.

There is a substantial number of renamings classified as *narrow* and *broaden meaning*. For example, the method `testEJB3RemoteAccess` of JBoss was renamed to `testRemoteAccess` to emphasize a more general behavior of the involved entity. A similar example is represented by the method `getServletRequest` of Tomcat that was renamed to `getRequest`. There are also cases where the identifier was made more specific. For example, the method `isRemoteInvocationExecutedInNewThread` of JBoss was renamed to `isRemoteAsyncInvocationExecutedInNewThread` to highlight that the remote invocation is asynchronous. A similar example is represented by the renaming `type` → `authType` in Tomcat.

There is also a high number of renamings that *add* or *remove meaning*. An example of adding a meaning is `delete` → `removeFromDiagram`, whereas an example of remove meaning is `addRecord` → `add`. Although these two kinds of renamings cover about half of the renamings identified by REPENT, it is worthwhile to point out that the interpretation can be subjective. Some of the examples may be classified differently by different people, *e.g., narrow meaning* rather than *add meaning*.

Finally, 5% of the renamings contain no semantic change, *i.e.*, are classified as *none*.

Our qualitative analysis confirms that in many renamings the goal of developers when performing renamings is to improve the comprehensibility of identifiers. We observed that most of these renamings are performed to increase the consistency between the name of an entity and its functionality, or between an identifier and other identifiers. This goes along with the high number of survey participants who would definitely rename an entity when the name and functionality are inconsistent (66%). Specifically, analyzed renamings aimed at improving the consistency with the existing code. For example, the method `isChildOf` in Eclipse-JDT was renamed to `isDescendantOf` as its functionality considers all super types, rather than the direct parent only. Sometimes developers rename identifiers to reflect new functionality represented by an entity. For example, field `typeMapping` in JBoss was renamed to `datasourceMapping`. The analysis of the log message confirmed that name changed to reflect the new functionality: *"Changed type-mapping to datasource-mapping as is required by new dtd."* Another example is the parameter `principal` in JBoss, renamed to `authPrincipal`. Here the renaming was a result of a bug fixing (*"incorrect principal used"*).

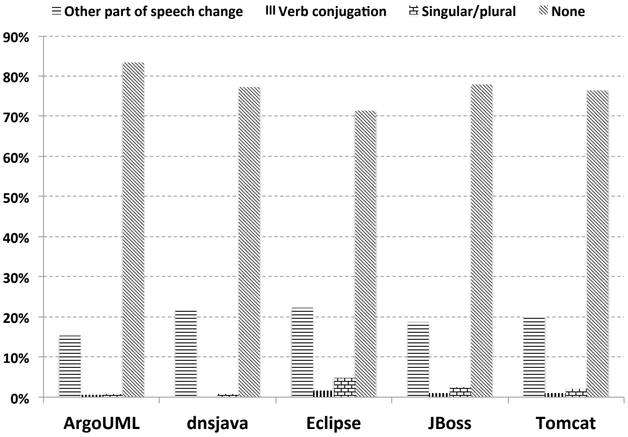


Fig. 14. Proportion of grammar changes identified by REPENT.

RQ3 conclusion: Renamings rarely preserve the meaning of identifiers (less than 3%). Slightly more often, the meaning is narrowed (6%), or broadened (5%). Moreover, renamings with no semantic changes are rare (5%). Most often, renamings change (33%), add (25%), or remove (21%) a meaning.

6.2.5 RQ4: What kinds of grammar changes occur in identifiers when they are renamed?

Table 13 and Fig. 14 show the proportion of the grammar changes in the five programs.

76% of the classified renamings do not involve a part of speech change, *i.e.*, are classified as *none* in the *grammar change* dimension. When there is a part of speech change however, only 5% of the changes involve a change in *verb conjugation*; 13% involve a *singular/plural* change. The majority of the renamings, *i.e.*, 83%, involve *other part of speech* changes.

One good reason for developers to change *singular* to *plural* and vice versa is to align an identifier with the entity (or collection of entities) to which it refers. For example, a field or a local variable of a collection type (*e.g.*, `ArrayList`) should have a plural name, whereas atomic entities should have, in general, a

TABLE 13
Grammar change renamings as classified by REPENT.

	Singular/ Plural	Verb conjugation	Other part of speech change	None	Total
ArgoUML	31	22	608	3,322	3,983
dnsjava	8	0	241	846	1,095
Eclipse	602	198	2,785	8,951	12,536
JBoss	337	125	2,531	10,589	13,582
Tomcat	51	25	479	1,805	2,360
Total	**1,029**	**370**	**6,644**	**25,513**	**33,556**

singular name. Such inconsistencies have been previously studied and denoted as *linguistic antipatterns* (LAs) [5] *i.e., recurring poor practices in the naming, documentation, and choice of identifiers in the implementation of an entity, thus possibly impairing program understanding.* That is, a method name containing only singular nouns but returning a collection is a LA of kind "Expecting but not getting a single instance". Similarly, method names containing plural nouns but returning a single object are LA of kind "Expecting but not getting a collection". Similar considerations can be made for fields. In our study, one example of renaming aimed at removing a LA occurred in JBoss, class `BasicMBeanRegistry`, where a local variable named `descriptors` was renamed to `descriptor` because its type changed from `Descriptor[]` to `Descriptor`. A similar case occurred in ArgoUML (class `LabelledLayout`) where a local variable of type `int` named `unknownHeights` was renamed to `unknownHeightCount`, the new name being consistent with the type. In this case, the program only keeps the count of heights rather than the list of heights, as suggested by the old name. In Tomcat, a method of class `RealmBase` named `findSecurityConstraint` was renamed to `findSecurityConstraints` and its return type changed from `SecurityConstraint` to `SecurityConstraint[]`.

Examples of *other part of speech changes* include the parameter renaming `localDeclaration` → `location` and the method renaming `deployOnMember` → `doDeployment`. There were renamings where although the part of speech changes, the role played by the renamed term remains the same. Examples are the method renaming `getTreeCache` → `getClusteredCache` and the field renaming `_multiPane` → `_editorPane` where although the part of speech of the renamed terms changed from noun to adjective and from adjective to noun respectively, the role played by the terms before and after the renaming is the same, *i.e.,* modifier of the term after, *i.e.,* `Cache` and `Pane` respectively.

REPENT reports 370 *verb conjugation* changes. In 32% of such changes the term `is` is renamed to `get` or `has` (or vice versa). The part of speech that the tagger assigns to the verb `is` and `has` is different from the part of speech of verb `get`. When using automatic code generator tools (such as the Eclipse IDE) for generating getters and setters, the names of getter methods returning a boolean value start with `is`. This is also the recommendation of JAVA/J2EE naming conventions. This could be a possible reason to rename getter methods of Boolean fields to start with `is`. One such example is the renaming `getValidProject` → `isValidProject`, where the return type of both methods is Boolean. The rest of verb conjugation changes are change in the verb tense (past to present or vice versa) or changes of a verb to gerund. Examples include method renamings such as `methodNeedingAbstractModifier` → `methodNeedBody` and `isOverridden Method` → `areOverriddenMethods`.

RQ4 conclusion: With respect to the grammar changes, 76% of the classified renamings did not involve a part of speech change. Of the 24% of the renamings involving a part of speech change, a small proportion involved a verb conjugation change (5%); 13% involved changes in nouns (singular/plural); 83% involved other part of speech changes.

7 Threats to Validity

This section discusses the threats to validity that can affect the studies performed in this paper. We discuss the most important threats that can affect this kind of study, *i.e., construct validity*, *conclusion validity*, and *external validity*. When needed, we divide the discussion of each kind of threat in issues related to the detection performance study and issues related to the renaming classification study.

Construct validity threats concern the relationship between theory and observation. As for the renaming detection study, construct validity threats are due to the detection of renamed files, the estimation of precision, and recall. For programs using SVN versioning system, when files are not explicitly renamed we compare deleted versus added files for two consecutive revisions. We use the Unix *diff* algorithm to compare the number of changed lines between all possible combinations of added and deleted files. We select the best possible combination (*i.e.,* smallest number of lines changed) if it does not exceed a relative threshold of 60%. The value for the threshold is estimated based on the central tendency of explicitly renamed files as logged by the versioning system files. For CVS, we first group files based on the commit date, log message, and committer ID [58], and then apply the same heuristic used for SVN, considering as deleted files the files that appear for the first time in the "Attic" directory.

As for precision, the manual validation could be affected by subjectiveness or human error. Specifically, regarding the classification, we may be affected by the lack of domain knowledge—i.e., the original developers of the five programs may have classified differently some of the renamings. To mitigate those threats, the validation was performed by two persons independently and, in case of different classification, the renaming was discussed, and a third person was also asked to perform the classification. As for the recall, we are aware that the sample of documented renamings may not be fully representative of the entire set of renamings performed in a project. First, this happens because developers do not always document renamings in commit notes, especially if they are performed together with other changes. Second, most of the documented renamings are related to types and thus to constructors, *i.e.*, to entities whose names can impact on other developers' activities.

For what concerns the classification study, construct validity threats are related to (i) the precision and recall of the set of detected renamings on which the classification is performed, and (ii) the accuracy of the automatic classification. Concerning the former, results of the study reported in Section 5 provide an indication of such precision and recall. Concerning the latter, we performed—using a process similar to the one described above—a manual validation of a sample of the classified renamings, to provide an idea of how accurate such a classification is.

Internal validity threats are related to factors, internal to our study, that can affect our results. As for the renaming detection, such a threat is mainly due to the calibration of thresholds. Indeed, different calibration could have produced different results, and also indirectly affected the subsequent renaming classification study. Appendix B explains how thresholds have been empirically determined. Clearly, the calibration has been performed based on a thorough validation on a sample set of detected renamings of Tomcat, when using low thresholds. The use of data from one of the projects to calibrate thresholds was also used in other studies [6], [30], [52]. However, this does not guarantee that the choice is optimal for the other projects.

Conclusion validity threats concern the relationship between the experimentation and the outcome. Our study is an exploratory study in which we do not make use of statistical tests to reject specific hypotheses. The only issue related to conclusion validity is the representativeness of the sample used to validate the renaming detection precision and the classification accuracy. In the first case, *i.e.*, to evaluate the detection accuracy, we performed for each program a stratified random sampling across the kinds of entities considering a confidence level of 95% and a confidence interval of at least $\pm 5\%$. In the second case we performed a random sampling for all dimensions and levels of the taxonomy considering a confidence level of 95% and a confidence interval of at least $\pm 10\%$.

External validity threats concern the generalizability of our results. Both the evaluation of the renaming detection approach and the exploratory study were conducted on data from a subset of the evolution history of five Java open-source projects. Although we have chosen a pretty variegated set of projects—belonging to different domains and development organization, and having different size—it could happen that replicating the study on other projects could lead to different results, *e.g.*, different performances of the renaming detection tool, or different distribution of the detected renamings with respect to the proposed taxonomy. A different matter is the application of the proposed approach on programming languages different from Java. In principle, the proposed approach is applicable to other languages, but some adaptation may be needed. For example, differently from Java identifiers, C/C++ identifiers are more likely to contain abbreviations [24], [38], and would rarely use camel case or other explicit term separators. As explained in Section 4.3.1, this would require the use of appropriate identifier splitting or normalization approaches [13], [18], [24], [32].

8 Related Work

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

8.1 Role of Identifiers in Software Quality

There is quite a consensus among researchers [12], [15], [18], [33] 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 [4], [39], measure conceptual cohesion and coupling [41], [46], and, in general, high quality identifiers are considered an asset for source code understandability and maintainability [34], [35], [51].

As suggested by Deissenbock and Pizka [15], identifiers should be consistent and concise. Unfortunately, verifying consistency and conciseness is a difficult task and thus approaches have been developed to detect consistency and conciseness violations by identifying usages of synonyms and holonyms [33]. 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 Lawrie *et al.* [33] that synonyms can indeed affect consistency. However, we also believe that renaming towards synonyms or towards other semantically-related terms—such as hypernyms, hyponyms, and antonyms—should be investigated as they likely point to program understanding issues.

8.2 Analysis of Changes and Refactoring

Several authors have proposed automated approaches to detect different kinds of refactorings.

At design level, Xing *et al.* [55] propose UMLDIFF to detect refactorings. UMLDIFF works with class diagrams; it inputs two class diagrams and it produces as output an XML design differencing file. By querying such a XML file, it would be possible to detect simple (*e.g.*, rename class/method/field, pull-up/push-down method/field) and composite refactoring actions (*e.g.*, replace inheritance with delegation).

Demeyer *et al.* [16] detect object-oriented refactorings based on a set of heuristics defined in terms of changes of object-oriented metrics measuring two successive software versions of Smalltalk programs.

Dig *et al.* [17] propose REFACTORING CRAWLER for detecting sequences of refactorings between consecutive versions of Java programs. REFACTORING CRAWLER identifies seven types of refactoring. Among others, they detect—as does REPENT—-package, class, and method renaming. The detection algorithm consists of a fast syntactic analyzer followed by a more computationally intensive semantic analyzer. The syntactic analyzer finds similar text fragments between two versions of source code based on Shingle encoding [9] as candidates of refactorings. The semantic analyzer further filters the candidate pairs to reduce false positives.

Weissgerber and Diehl propose a signature-based approach to identify refactorings [54]. The approach starts with collecting and pre-processing data from the version control system. Next, it identifies added and removed entities (classes, interfaces, methods, and fields) in each transaction. Those entities are then compared based on their signatures and potential refactorings are identified. The approach then ranks and filters potential refactorings based on the similarity of the entity body in the old and new version. To measure the similarity between the two versions, the approach first tests for string equality. Then if it fails, the approach uses the result of a token-based code clone detection algorithm, *i.e.*, CCFINDER [28]. Among the detected refactorings, the approach detects "Rename Method" and "Rename Class" refactorings, as REPENT does.

Prete *et al.* [47] propose REF-FINDER as a way to detect atomic refactorings and then based on logic templates reconstruct more complex refactorings (such as extract method). REF-FINDER detects method renamings based on method body similarity.

It is important to point out that the approaches described above have been conceived to detect refactorings in general. They detect only a subset of the renamings detected by REPENT, and do not perform a classification of the detected renamings.

Malpohl *et al.* [40] propose 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. To increase accuracy, variable types and references are compared for matching the identifiers. Malpohl *et al.* evaluated the technique on two consecutive versions of the tool itself. They report a 100% precision rate for of the 77 analyzed file pairs. We share with Malpohl *et al.* the general idea as well as the use of data-flow analysis in the renaming detection process. However, our approach for the detection of renamings is substantially different. Specifically, it is a multi-stage approach, in which an initial filtering localizing changes based on differencing analysis is then followed by a data-flow analysis on candidate renamings, aimed at filtering out false positives. This allows us a better scalability, and hence the ability to analyze the evolution of large projects such as JBoss or Eclipse-JDT.

Neamtiu *et al.* [44] propose an approach for understanding code evolution using AST matching. They analyze open source programs written in C and provide a release digest that summarized changes between two subsequent releases of a file. Renamings of types and variables are part of the changes detected by the authors. Those renamings correspond to our class, field, local variable, and parameter renamings. The approach does not handle method renamings, nor local variable and parameter renamings when the latter are in a renamed method because the approach is based on the hypothesis that in C functions are relatively stable over time. Thus, to detect local variable and parameter renamings, Neamtiu *et al.* compare the ASTs of two methods with the same name. Such an assumption would be unrealistic for Java programs. As our results show, method renamings represent 26% of the renamings for the 5 programs.

Fluri *et al.* [20] propose a tree differencing algorithm, CHANGEDISTILLER, for extracting the changes from two consecutive versions of Java files. Renaming is a type of change that CHANGEDISTILLER can detect together with many others. The algorithm compares the ASTs of the files and computes the edit operations to transform the AST of the old version of a file to the AST of the new version of the same file. Fluri *et al.* used bi-gram string similarity for calculating the similarity between two leaf nodes (*i.e.*, identifier names). Thus, the detection of renamings is based on

TABLE 14
Accuracy of REPENT and DiffCat on a random sample of revisions.

	Precision		TP / Detected		FN	
	REPENT	DiffCat	REPENT	DiffCat	REPENT	DiffCat
dnsjava	99%	96%	104 / 105	99 / 103	16	21
JBoss	81%	72%	108 / 133	77 / 107	33	63

the declaration, whereas in our case we consider def-uses when available. As in our case, the detection of renamings depends on the pre-established thresholds. To evaluate their approach, Fluri *et al.* built a benchmark that consists of 1,064 manually classified changes of eight methods extracted from three open source programs. Only four of the classified changes are renamings, specifically one method and three parameters.

Kawrykow and Robillard [29] measured the impact of non-essential differences on approaches aimed at detecting change couplings based on association rule discovery [58]. Kawrykow and Robillard treated the renaming of an entity as an "essential" change, while the updated (*i.e.*, impacted) statements of this renaming are considered as "non-essential". Although the end goal is different from ours, detecting renamings is part of both approaches. To detect renamings Kawrykow and Robillard propose DiffCat, which is built on the approach of Fluri *et al.* [20]. That is, they used ChangeDistiller to detect method and field renamings. However, they further enhanced it to also detect class, local variable, and additional field renamings. Since the goal of DiffCat is different from ours, it favors precision at the expense of recall. We compared REPENT to DiffCat on renamings detected in a random sample of revisions of dnsjava and JBoss. For each randomly selected revision, we applied both tools and manually validated the detected renamings. We stopped sampling once both approaches reached a total of 100 renamings. Table 14 shows the results of the comparison[14]. The last column of Table 14 reports the false negatives (FN), *i.e.*, the number of true renamings that each approach does not detect. Overall, REPENT outperforms DiffCat in terms of precision and number of detected true renamings.

REPENT uses file context *diff* to reduce the search space of entities involved in potential renamings, the actual validation of renamings is mainly based on the def-use analysis or, when not available, on the similarity between declarations. File context *diff* can be replaced with any other differencing tool including ChangeDistiller. However, the advantage of the file context *diff* is its speed and a simple comparison between file context *diff* and ChangeDistiller showed that both tools are sensitive to cases where chunks of code are moved within a file. In addition, the comparison with DiffCat, which is built on ChangeDistiller, shows that REPENT is more accurate in terms of detected renamings.

We share with all of the above approaches their general ideas and goal. We also use parsing and differencing technologies, though in a different combination, to achieve 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, and a classifier that—by relying on WordNet and on the Stanford NLP—classified renamings according to the proposed taxonomy. Thus REPENT detects finer-grain details about renamings, such as the grammatical renaming type or the semantic type. Furthermore, our approach does not require a compilable program to work.

9 Conclusion and Future Work

Identifier renaming is considered by developers as an important and non-trivial task in the context of software evolution. We conducted a survey with 71 industrial and open-source developers and show that, as part of the evolution of software programs, developers rename identifiers to improve the quality of the source code lexicon and its consistency with the program functionality. Despite the importance of renaming and the associated with it cost (92% of the surveyed developers do not consider renaming as straightforward), renamings are hardly ever documented (1% of the renamings in the five programs that we studied) hence the need for an automatic documentation of renamings that 52% of the surveyed developers consider useful.

In this paper we propose REPENT (REnaming Program ENTities)—an approach to automatically document, *i.e.*, detect and classify, identifier renamings between different versions of a Java program. For detecting renamings, REPENT first reduces the search space by identifying changes from mapped source code lines between versions of a program resulting in candidate renamings, after which it performs data flow analysis on the entities involved in the candidate renamings to filter out false positives. We analyzed renamings detected by REPENT in the evolution history of five open-source programs (ArgoUML, dnsjava, Eclipse-JDT, JBoss, and Tomcat), and reported a precision of 88% and a recall—with respect to the documented renamings—of 92%. By combining an ontological database (WordNet) with a natural language parser (Stanford NLP) REPENT classifies renamings according to the different dimensions of our taxonomy, and specifically (i) the kind of entity being renamed, (ii) the form of the renaming, (iii) semantic change, and (iv) grammatical change. By

[14] The detected and validated renamings can be found in the replication package.

relying on REPENT, we conducted an exploratory study—on the five Java programs used to evaluate the performances—aimed at determining how developers rename identifiers according to the proposed taxonomy.

As pointed out above, REPENT relies on external tools such as the Unix diff, WordNet, and Stanford NLP, which turned out to suffice for our needs. However, future developent activities could easily extend REPENT by replacing such tools with alternative and possibly more accurate ones.

We conjecture that renamings documented and classified by REPENT can be used as a base towards building a renaming recommender system. This also reflect results of our survey, where 68% of the surveyed developers indicated the usefulness of automatic recommendations for renaming, provided that such recommendations are non-intrusive and offer reliable suggestions.

More specifically, REPENT can be the core component of renaming recommenders that, by learning from past renamings, could automatically point out inconsistencies to developers—*e.g.*, linguistic antipatterns [5]—to proactively suggest how to rename identifiers to improve source code comprehensibility as well as pointing to past renamings conflicting with the ongoing renaming activity. Such foreseeable recommenders would for example be useful in the following situation. In revision 67429 of JBoss, method `deploy` was renamed to `internalDeploy` in five different classes. In the same revision, in three of those classes method `unDeploy` was renamed to `internalUnDepoly`. The same set of renamings occurred at a later stage (revision 79147) for a different class. Hence, documenting the renamings in revision 67429 and learning from them would have facilitated the work of developers in later revisions, when creating an entity or renaming it, by pointing to names used in a similar context. If such a recommendation is not accepted by the developer, it will still be beneficial as it will be clear that such contrast is deliberate and it is performed with the developer's full awareness, thus the rationale behind it must be explicitly documented for future evolution.

Besides building a renaming recommender, work-in-progress also aims at supporting programming languages other than Java, as for example scripting languages like PHP.

Appendix A
Survey

This appendix reports detailed results of the survey illustrated in Section 2.

First, we report information about participants' background. In particular, Fig. 15 shows statistics regarding the native language of the participants, whereas Fig. 16 reports their years of experience in industrial and open-source software development.

Fig. 17 reports how often developers rename. Only 14% of participants rename rarely (up to once per month): 46% rename occasionally (a few times per month) while 18% rename frequently (a few times per week) and 21% rename very frequently (almost every day).

Fig. 18 indicates activities during which developers rename. Note that a participant may select more than one activity, thus the sum of the percentages is above 100%. Participants rarely perform renaming as a standalone activity (17%). Often, they rename when performing other refactorings (90%), changing the functionality (89%), adding new functionality (65%), understanding code (51%), or fixing a bug (42%).

Fig. 19 provides insights about the opinion of participants about the cost of renaming. 35% of participants consider that renaming requires time and effort (at least in most cases); 32% consider that the cost of renaming depends on the particular case; 32% consider renaming to be straightforward (at least in most cases). Note that the sum of the above is 99% due to rounding errors.

Fig. 20 reports results on the use of tool support for renaming. The majority of the participants (72%) use automatic tool support to perform renaming. There are however participants that rename manually (20%) and participants that perform both, manual and automatic renaming (8%).

We asked participants to share the reasons for which they recall having decided not to rename an entity; results are shown in Fig. 21. 52% of the participants recall the reason to be the potential impact on other systems. 35% recall that the renaming was too risky, *i.e.*, it might have introduced a bug. 25% of the participants answered that the high impact of the renaming on the system was the show-stopper and finally, 25% recall deciding not to rename because of the high effort required.

We also asked participants whether a set of predefined factors would impact the decision to undertake a renaming. Results are not surprising (Fig. 22). The majority of participants consider important all those factors. However, the factor that is worth highlighting here is the impact on other projects—69% of participants say that this would definitely impact their decision.

Fig. 23 shows when developers feel the need to rename. As expected, the majority (66%) of participants clearly state that they will definitely rename an entity

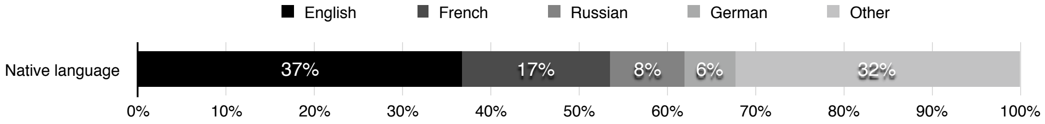


Fig. 15. Native language of the participants.

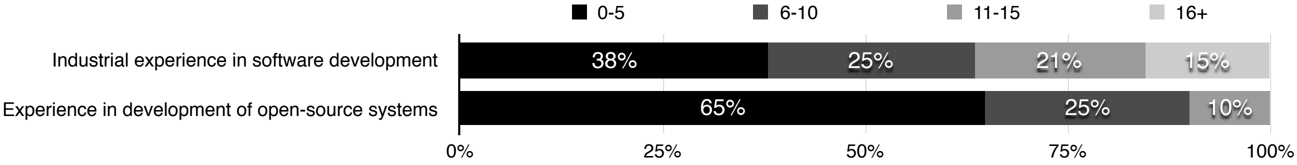


Fig. 16. Experience of the participants in software development.

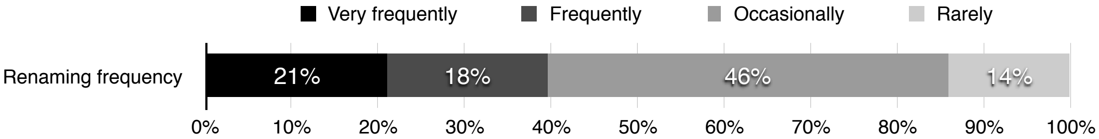


Fig. 17. How often do developers rename?

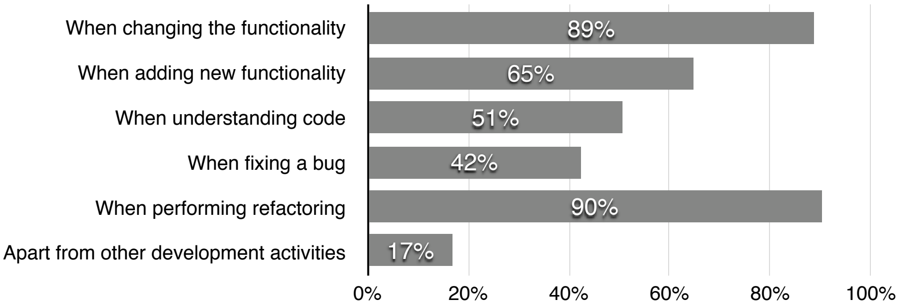


Fig. 18. Activities accompanying renaming.

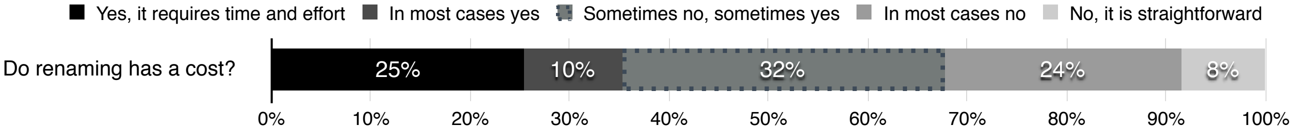


Fig. 19. Developers' opinion on cost of renaming.

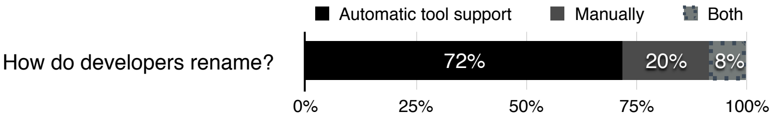


Fig. 20. How do developers rename?

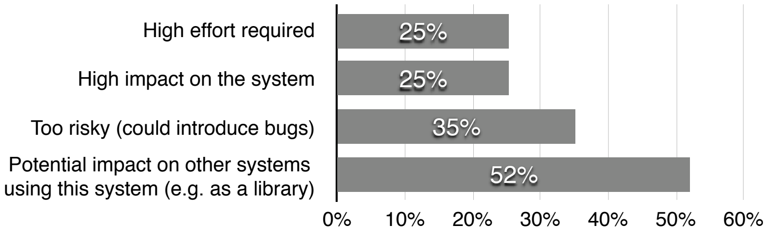


Fig. 21. Reasons for which developers already postponed or canceled a renaming.

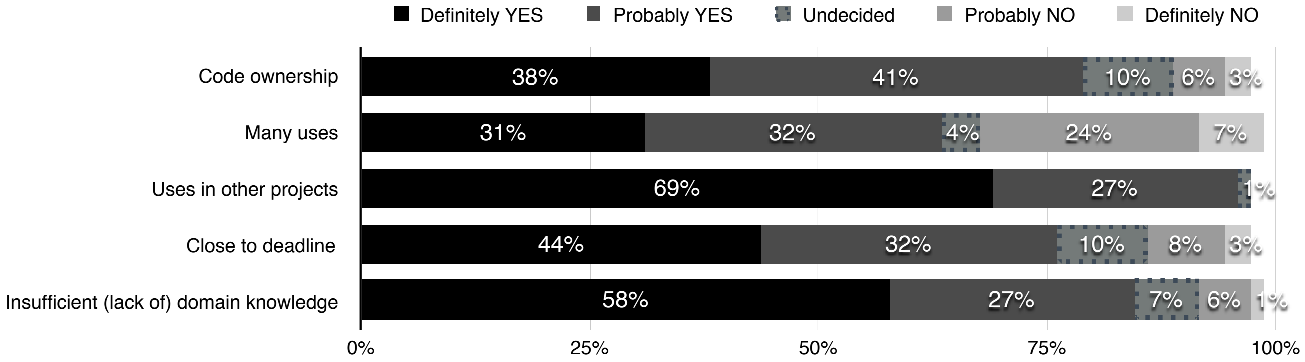


Fig. 22. Factors impacting developers decision to undertake a renaming.

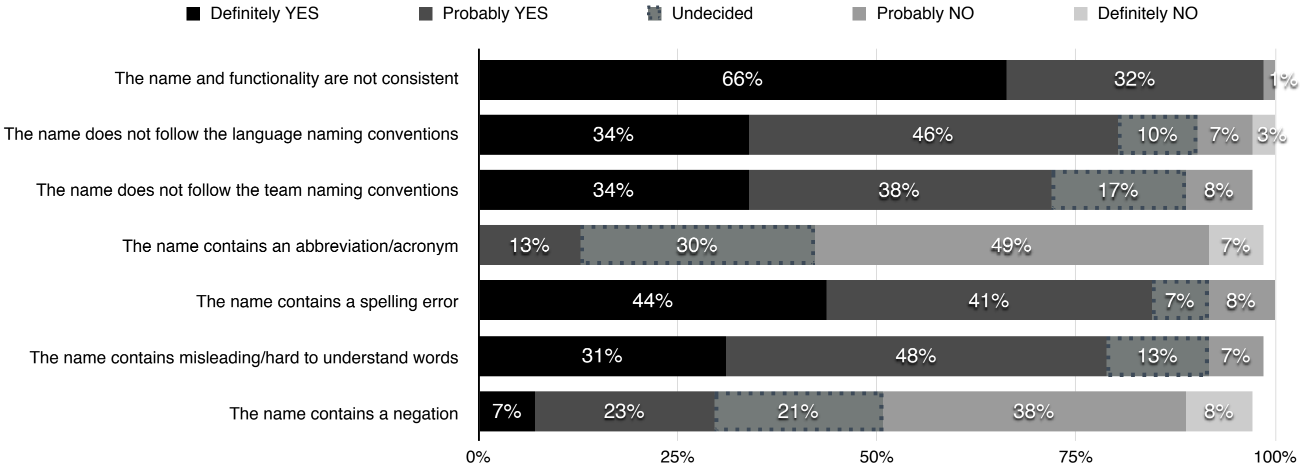


Fig. 23. When will developers rename?

if its name is not consistent with its functionality. They made less strong statements about naming conventions, spelling errors, and hard to understand words, but still the majority of participants report that they will probably rename in such cases. Surprisingly, only 13% of participants will probably rename if an entity contains an abbreviation—the majority of participants (56%) will not rename. Finally, when the name of an entity contains a negation, *e.g.*, `notOpen`, 30% of the participants will rename, while 46% will not.

We asked participants whether they consider useful automatically documenting renaming; results are shown in Fig. 24. The majority (52%) are positive. 33% of participants are negative about automatic documentation. The remaining participants did not provide their opinion.

Fig. 25 reports participants' opinion on renamings that are useful to document. More specifically, developers see the usefulness of automatically documenting renamings of public APIs, *i.e.*, classes and methods, and renamings concerning the meaning of the name—renaming towards opposite meaning, towards unrelated words, towards more general/specific names, adding/removing meaning. Surprisingly, the percentage of participants that see a benefit from automatic renaming towards synonyms is lower—36%. Similarly, but not surprisingly, a small number of participants see a benefit from automatically documenting renaming of entities with local scope, renamings towards abbreviations/expansions, and spelling errors.

The majority (68%) of participants see a benefit of automatic recommendations for renaming (Fig. 26) provided that such recommendations are non-intrusive and offer reliable suggestions. Fig. 26 reports developers' opinion on the usefulness of recommending renamings. Finally, participants see the benefit of recommendations regarding the majority of renamings (Fig. 27).

Appendix B
Setting the Thresholds

B.1 Thresholds for detecting renamings

REPENT uses three sets of thresholds, where each threshold varies in the range $[0, 1]$. Specifically, the REPENT thresholds are described as follows:

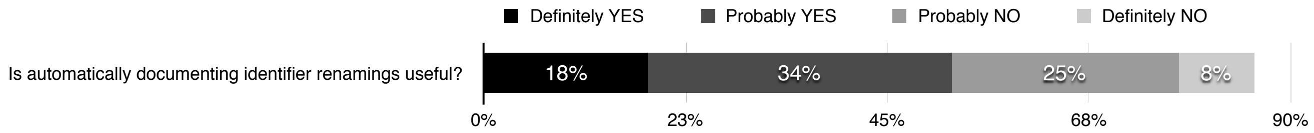


Fig. 24. Developers' opinion on the usefulness of documenting renamings.

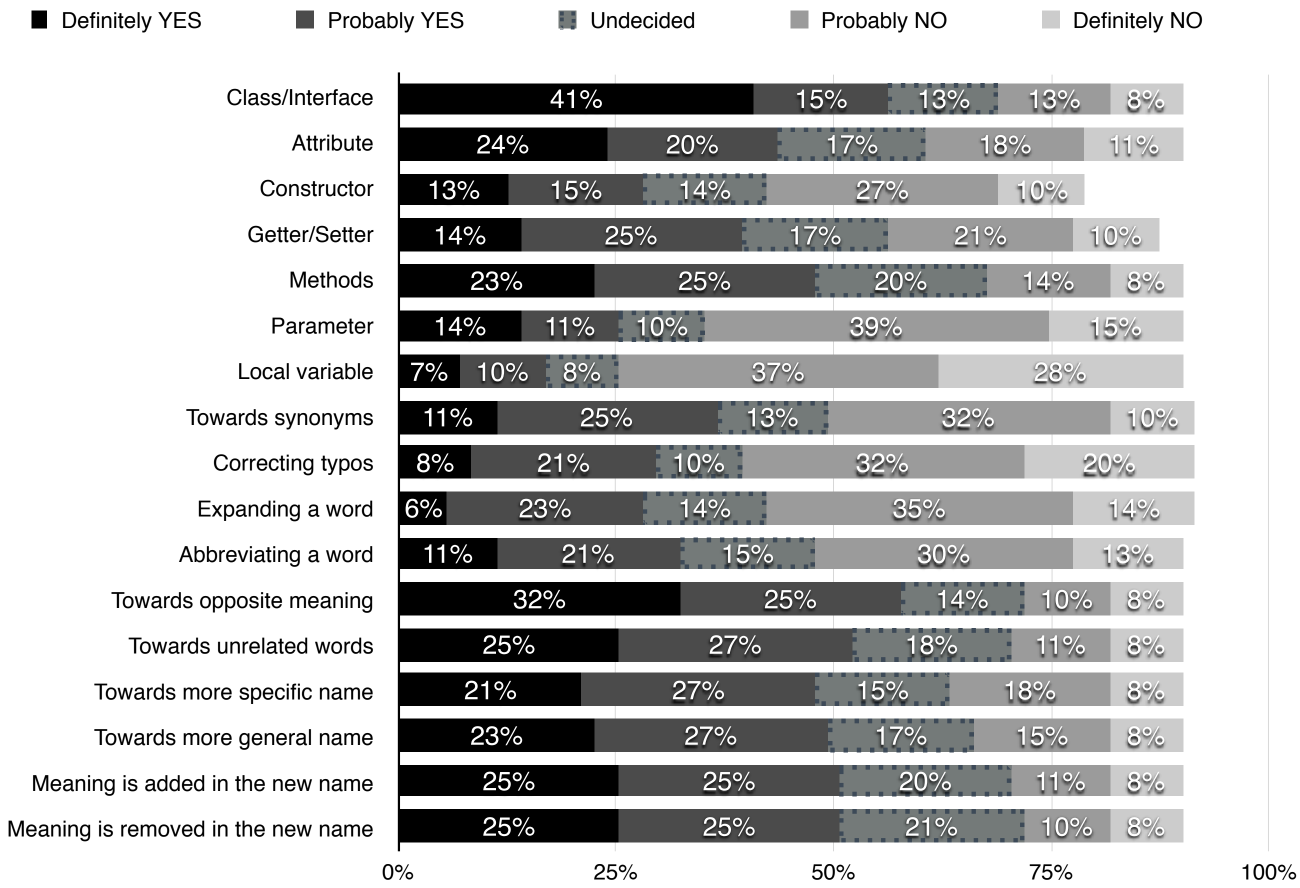


Fig. 25. Developers' opinion on renamings that are useful to document.

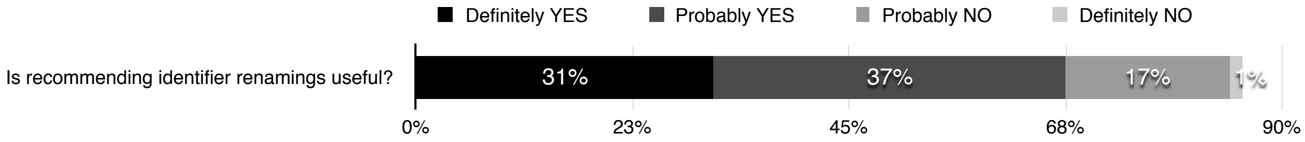


Fig. 26. Developers' opinion on the usefulness of recommending renamings.

- **Statement Similarity Threshold (SST):** used to match def-uses of the mapped entities.
- **Number of Matched Statements Threshold (NST):** used to decide if the mapped entities have a sufficiently high number of matched def-uses statements, to support the evidence of a real renaming.
- **Declaration Similarity Threshold (DST):** used to match the declaration of two mapped entities, whenever one or both entities do not have uses.

While DST has been set to a constant value not depending on the kind of entity, *SST* and *NST* require different calibrations to be able to work effectively on different kinds of entities, in order to balance false positives as well as false negatives. The rationale is that for different entities (class names, method names, etc.) the syntax and frequency of statements where def-uses occur may be quite different.

We calibrate the thresholds by varying NST between zero and one, with step 0.1; for each fixed value of *NST* (*e.g.*, 0.4), we made *SST* vary (between zero and one with step 0.1).

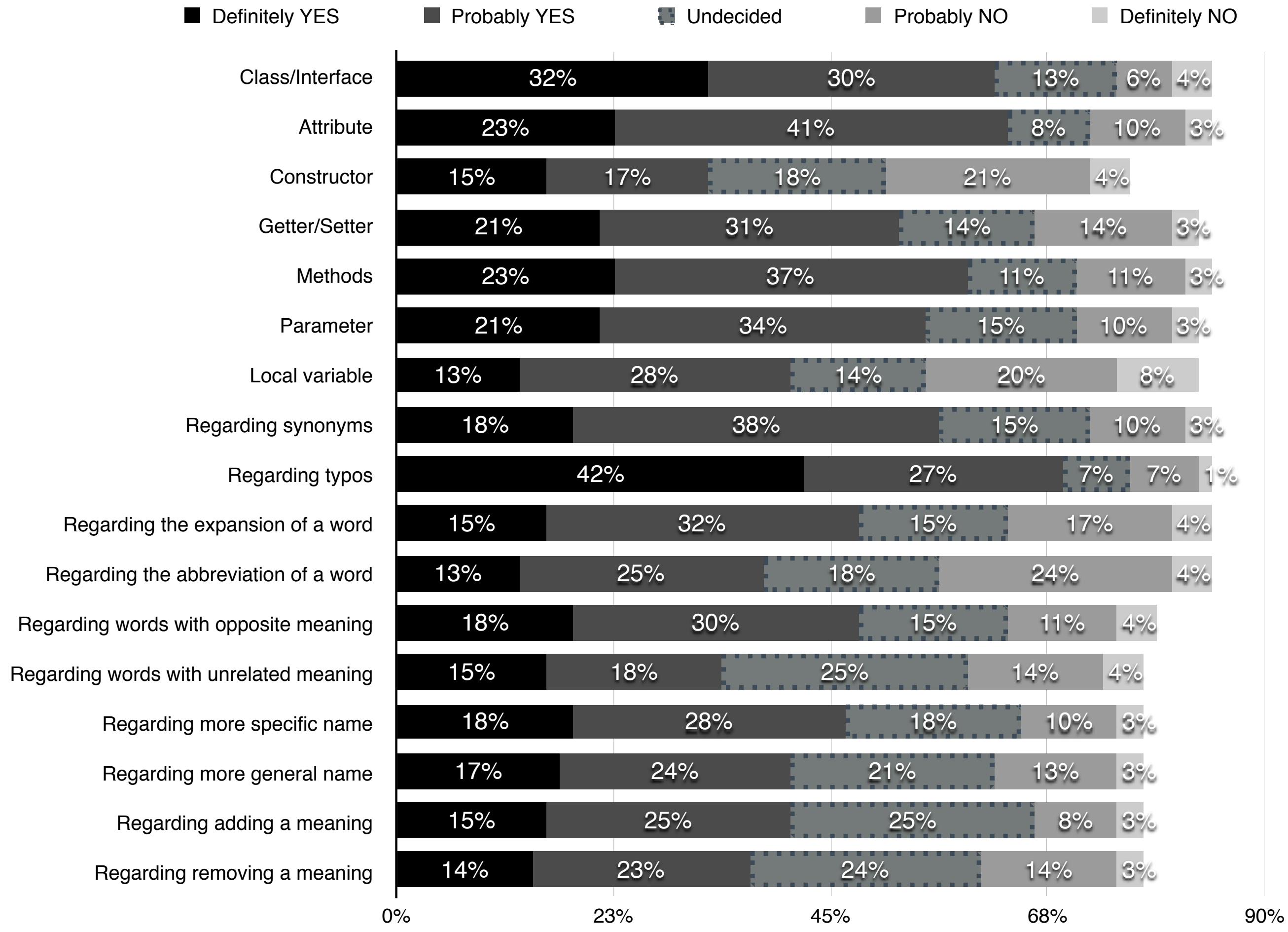


Fig. 27. Developers' opinion on renamings that are useful to recommend.

TABLE 15
Thresholds chosen for the study as well as corresponding TPR and FPR on the calibration set.

Kind of entity	SST	NST	DST	TPR		FPR		Precision
Type	0.8	0.3	0.7	89%	(17/19)	0%	(0/1)	100%
Constructor	0.6	0.8	0.7	100%	(7/7)	0%	(0/4)	100%
Field	0.8	0.4	0.7	86%	(262/303)	12%	(20/163)	93%
Method	0.8	0.3	0.7	95%	(349/368)	18%	(11/61)	97%
Getter/Setter	0.8	0.2	0.7	93%	(250/270)	18%	(22/122)	92%
Parameter	0.7	0.5	0.7	64%	(214/334)	36%	(48/132)	82%
Local variable	0.8	0.3	0.7	77%	(172/224)	4%	(11/257)	94%
Overall				83%	(1271/1525)	15%	(112/740)	92%

We used Tomcat as a calibration data set, *i.e.*, we used the oracle of 724 renamings detected in our previous study [19] and manually classified. In addition, to better distinguish how REPENT performs with different thresholds, we computed the set of renamings that change when thresholds vary—we computed the union and intersection of the detected renamings for all combinations of the different values of *NST* and *SST*. We then randomly sampled renamings from the complement of the intersection until reaching an oracle that sufficiently discriminates the different thresholds—we stopped when the oracle reached 2,265 renamings. Next, we computed the True Positive Rate (TPR) and the False Positive Rate (FPR) for each entity kind on the calibration set. TPR and FPR were used to generate a family of Receiver Operating Characteristic (ROC) curves. ROC curves plot the TPR over the FPR at various threshold settings. Notice that TPR is equivalent to recall. The relation between FPR (TPR) and precision is more complex as precision is the ratio of true positives over the sum of true positives and false positives; however, reducing FPR increases the number of correctly classified items, and thus it improves precision.

Fig. 28 shows a subset of ROC family computed for class FielD (FD); top right curve (symbol +) was obtained setting *NST*= 0.1, while the bottom left curve (symbol ▲) corresponds to NST=0.9. REPENT

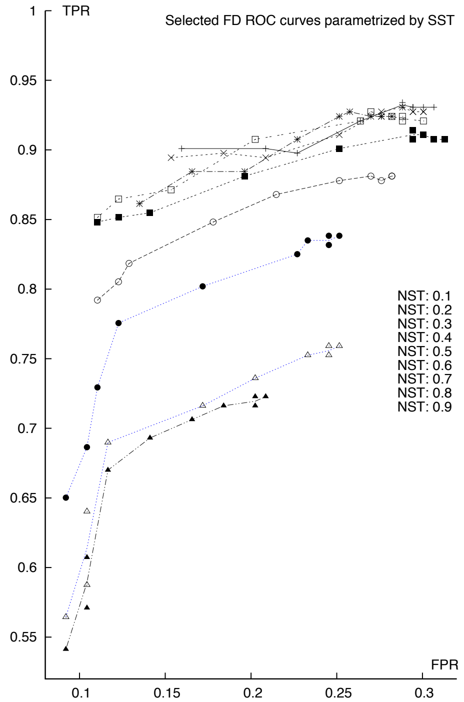


Fig. 28. REPENT class FielD (FD) ROC curves as functions of SST and NST.

works on the ROC curve with *NST*= 0.4 (symbol ⊡). The *SST* value was fixed at 0.8 which gives (on the calibration set) TPR of 86% and FPR of 12%. As shown in Fig. 28, these two values are the threshold values giving (on our FD calibration set) the highest TPR with a reasonably low FPR for FD; no other ROC curve has a lower FPR with a higher (or equal) TPR. For example, the topmost-right ROC curve (denoted with +) indicates a higher TPR at a price of a higher FPR. In essence, depending on the goal to be achieved, REPENT can be calibrated to favor low FPR, high TPR or a compromise between the two. This latter choice was used to select, via the same analysis process, for each entity kind *SST* and *NST* values reported in Table 15.

B.2 Thresholds for classifying renamings

To classify renamings as spelling errors REPENT uses a threshold for the Levenshtein distance between the old and new name of an entity. Lower values for the Levenshtein distance threshold ensure a high precision in the classification, but also a higher number of false negatives. Table 16 shows the accuracy of the classification of renamings as spelling errors on Tomcat when the Levenshtein distance threshold varies from 1 to 5. A threshold of 1 ensures 0% FPR while failing to classify as spelling errors renamings such as `refeelReadBuffer` → `refillReadBuffer`. Increasing the threshold to 2 solves this issue while keeping a low FPR (4% on Tomcat). An example of misclassified renaming when the Levenshtein distance threshold is set to 2 is `class` → `clazz`.

TABLE 16
Accuracy of the classification of renamings as spelling errors using different Levenshtein distance thresholds on Tomcat.

Threshold value	TPR	FPR
1	86%	0%
2	100%	4%
3	100%	25%
4	100%	50%
5	100%	100%

Appendix C
Sampling for evaluation

C.1 Sampling for evaluating the detection

Table 17 reports the sample size and the number of detected renamings, overall and for each kind of entity (we highlight in bold face the size of the significant sample). For example, for ArgoUML (first system in Table 17) we sampled 352 renamings for validation out of the 3,994 detected renamings for that program.

C.2 Sampling for evaluating the classification

Tables 18 to 20 report the sample size and the number of correctly classified renamings for each dimension of taxonomy. We highlight in bold face the significant samples and the corresponding precision. For example, regarding *forms of renaming* (Table 18) we sampled 96 *simple* renamings and 93 of them are correctly classified thus resulting in a precision of 97%. We do not evaluate the accuracy of the classification for the entity kind dimension as it is correct by construction, *i.e.*, it is extracted when parsing the source code, and its only imprecision could also be due to mistakes in the Eclipse JDT parser we used.

References

[1] Abbott, R.J.: Program design by informal english descriptions. Commun. ACM **26**(11), 882–894 (1983)

[2] Abebe, S.L., Haiduc, S., Marcus, A., Tonella, P., Antoniol, G.: Analyzing the evolution of the source code vocabulary. In: Proceedings of the European Conference on Software Maintenance and Reengineering, pp. 189–198 (2009)

[3] Abebe, S.L., Tonella, P.: Natural language parsing of program element names for concept extraction. In: Proceedings of the International Conference on Program Comprehension (ICPC), pp. 156–159 (2010)

TABLE 17
Sample size to estimate the precision of REPENT.

	ArgoUML		dnsjava		Eclipse-JDT		JBoss		Tomcat		Overall	
Package	1 /	1	0 /	0	1 /	4	1 /	7	0 /	0	3 /	12
Type	2 /	18	17 /	67	5 /	180	18 /	656	9 /	70	51 /	991
Constructor	1 /	16	40 /	159	4 /	139	13 /	475	7 /	49	65 /	838
Field	190 /	2,156	15 /	59	58 /	1,945	49 /	1,808	67 /	498	379 /	6,466
other-MD	17 /	200	44 /	177	58 /	1,966	66 /	2,397	59 /	441	244 /	5,181
Getter/Setter	17 /	192	11 /	45	37 /	1,244	44 /	1,595	57 /	423	166 /	3,499
Parameter	61 /	696	126 /	506	96 /	3,226	94 /	3,419	75 /	561	452 /	8,408
Local variable	63 /	715	37 /	149	114 /	3,853	90 /	3,261	59 /	439	363 /	8,417
OVERALL	**352** /	3,994	**290** /	1162	**373** /	12,557	**375** /	13,618	**333** /	2,481	1,723 /	33,812

TABLE 18
Evaluation of classification for "Forms of renaming".

	ArgoUML		dnsjava		Eclipse-JDT		JBoss		Tomcat		Overall	
Simple	100%	(4/4)	100%	(1/1)	94%	(34/36)	100%	(43/43)	92%	(11/12)	**97%**	(93/**96**)
Complex	100%	(2/2)	100%	(2/2)	96%	(47/49)	94%	(32/34)	100%	(8/8)	**96%**	(91/**95**)
Formatting only	100%	(58/58)	100%	(6/6)	100%	(3/3)	100%	(24/24)	100%	(2/2)	**100%**	(93/**93**)
Term reordering	-		-		100%	(23/23)	100%	(18/18)	100%	(5/5)	**100%**	(46/**46**)
Overall	100%	(64/64)	100%	(9/9)	96%	(107/111)	98%	(117/119)	96%	(26/27)	98%	(323/330)

TABLE 19
Evaluation of classification for "Semantic changes".

		ArgoUML		dnsjava		Eclipse-JDT		JBoss		Tomcat		Overall	
Preserve	Synonym	50%	(1/2)	-		94%	(33/35)	97%	(36/37)	88%	(7/8)	**94%**	(77/**82**)
	Synonym phrase	-		-		0%	(0/1)	100%	(2/2)	-		**67%**	(2/**3**)
	Spelling error	-		100%	(1/1)	100%	(32/32)	94%	(32/34)	100%	(9/9)	**97%**	(74/**76**)
	Expansion	100%	(17/17)	100%	(1/1)	81%	(17/21)	90%	(19/21)	100%	(6/6)	**91%**	(60/**66**)
	Abbreviation	67%	(2/3)	-		93%	(13/14)	90%	(36/40)	100%	(9/9)	**91%**	(60/**66**)
	Overall	91%	(20/22)	100%	(2/2)	92%	(95/103)	93%	(125/134)	97%	(31/32)	93%	(273/293)
Change	Opposite	-		-		100%	(25/25)	88%	(14/16)	100%	(5/5)	**96%**	(44/**46**)
	Opposite phrase	0%	(0/1)	-		29%	(6/21)	44%	(8/18)	0%	(0/3)	**33%**	(14/**43**)
	Whole-part	-		-		-		-		100%	(2/2)	**100%**	(2/**2**)
	Whole-part phrase	-		-		-		-		-		-	
	Unrelated	67%	(4/6)	-		78%	(31/40)	79%	(33/42)	71%	(5/7)	**77%**	(73/**95**)
	Overall	57%	(4/7)	-		72%	(62/86)	72%	(55/76)	71%	(12/17)	72%	(133/186)
Narrow	Specialization	0%	(0/3)	100%	(1/1)	78%	(31/40)	35%	(11/31)	50%	(2/4)	**57%**	(45/**79**)
	Specialization phrase	100%	(3/3)	50%	(1/2)	70%	(28/40)	63%	(24/38)	56%	(5/9)	**66%**	(61/**92**)
	Overall	50%	(3/6)	67%	(2/3)	74%	(59/80)	51%	(35/69)	54%	(7/13)	62%	(106/171)
Broaden	Generalization	67%	(2/3)	-		85%	(40/47)	75%	(15/20)	63%	(5/8)	**79%**	(62/**78**)
	Generalization phrase	67%	(2/3)	-		65%	(33/51)	58%	(15/26)	36%	(4/11)	**59%**	(54/**91**)
	Overall	67%	(4/6)	-		74%	(73/98)	65%	(30/46)	47%	(9/19)	69%	(116/169)
Add		100%	(1/1)	-		84%	(47/56)	87%	(27/31)	43%	(3/7)	**82%**	(78/**95**)
Remove		100%	(3/3)	100%	(3/3)	94%	(46/49)	88%	(30/34)	67%	(4/6)	**91%**	(86/**95**)
None		100%	(40/40)	100%	(5/5)	100%	(16/16)	100%	(27/27)	100%	(5/5)	**100%**	(93/**93**)
Overall		88%	(75/85)	92%	(12/13)	82%	(398/488)	79%	(329/417)	72%	(71/99)	80%	(885/1102)

TABLE 20
Evaluation of classification for "Grammar changes".

			ArgoUML		dnsjava		Eclipse-JDT		JBoss		Tomcat		Overall	
Grammar change	Part of speech change	Singular/Plural	100%	(1/1)	-		100%	(57/57)	100%	(28/28)	100%	(2/2)	**100%**	(88/**88**)
		Verb conjugation	50%	(1/2)	-		79%	(23/29)	83%	(29/35)	70%	(7/10)	**79%**	(60/**76**)
		Other	40%	(2/5)	-		25%	(11/44)	9%	(3/33)	23%	(3/13)	**20%**	(19/**95**)
	None		100%	(26/26)	100%	(4/4)	100%	(24/24)	100%	(34/34)	100%	(8/8)	**100%**	(96/**96**)
Overall			88%	(30/34)	100%	(4/4)	75%	(115/154)	72%	(94/130)	61%	(20/33)	74%	(263/355)

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

[5] Arnaoudova, V., Di Penta, M., Antoniol, G., Guéhéneuc, Y.G.: A new family of software anti-patterns: Linguistic anti-patterns. In: Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR) (2013)

[6] Bavota, G., De Lucia, A., Oliveto, R.: Identifying extract class refactoring opportunities using structural and semantic cohesion measures. Journal of Systems and Software **84**, 397–414 (2011)

[7] Binkley, D., Hearn, M., Lawrie, D.: Improving identifier informativeness using part of speech information. In: Proceedings of the International Working Conference on Mining Software Repositories (2011)

[8] Black, T.R.: Doing Quantitative Research in the Social Sciences: An Integrated Approach to Research Design, Measurement and Statistics. Statistics Series. SAGE Publications (1999)

[9] Broder, A.: On the resemblance and containment of documents. In: Proceedings of the Compression and Complexity of Sequences, pp. 21–29 (1997)

[10] Bruegge, B., Dutoit, A.H.: Object-Oriented Software Engineering: Using UML, Patterns, and Java. Prentice Hall (2003)

[11] Capobianco, G., De Lucia, A., Oliveto, R., Panichella, A., Panichella, S.: Improving ir-based traceability recovery via noun-based indexing of software artifacts. Journal of Software: Evolution and Process (2013. To appear.)

[12] Caprile, B., Tonella, P.: Restructuring program identifier names. In: Proceedings of the International Conference on Software Maintenance, pp. 97–107 (2000)

[13] Corazza, A., Di Martino, S., Maggio, V.: Linsen: An approach to split identifiers and expand abbreviations with linear complexity. In: Proceedings of the International Conference on Software Maintenance, (ICSM 2012) (2012)

[14] Cormen, T.H., Leiserson, C.E., Rivest, R.L.: Introductions to Algorithms. MIT Press (1990)

[15] Deissenbock, F., Pizka, M.: Concise and consistent naming. In: Proceedings of the International Workshop on Program Comprehension (2005)

[16] Demeyer, S., Ducasse, S., Nierstrasz, O.: Finding refactorings via change metrics. In: Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications, pp. 166–177 (2000)

[17] Dig, D., Comertoglu, C., Marinov, D., Johnson, R.E.: Automated detection of refactorings in evolving components. In: Proceedings of the European Conference on Object-Oriented Programming, pp. 404–428 (2006)

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

[19] Eshkevari, L.M., Arnaoudova, V., Di Penta, M., Oliveto, R., Guéhéneuc, Y.G., Antoniol, G.: An exploratory study of identifier renamings. In: Proceedings of the 8th International Working Conference on Mining Software Repositories (MSR 2011), pp. 33–42 (2011)

[20] Fluri, B., Wuersch, M., PInzger, M., Gall, H.: Change distilling: Tree differencing for fine-grained source code change extraction. IEEE Trans. Softw. Eng. **33**(11), 725–743 (2007)

[21] Fowler, M.: Refactoring: Improving the design of existing code. Addison-Wesley (1999)

[22] Glaser, B.G.: Basics of grounded theory analysis. Sociology Press (1992)

[23] Groves, R.M., Fowler Jr., F.J., Couper, M.P., Lepkowski, J.M., Singer, E., Tourangeau, R.: Survey Methodology, 2nd edition. Wiley (2009)

[24] Guerrouj, L., Di Penta, M., Antoniol, G., Guéhéneuc, Y.G.: Tidier: An identifier splitting approach using speech recognition techniques. Journal of Software Maintenance - Research and Practice p. 31 (2011)

[25] Gupta, S., Malik, S., Pollock, L., Vijay-Shanker, K.: Part-of-speech tagging of program identifiers for improved text-based software engineering tool. In: Proceedings of the International Conference on Program Comprehension (ICPC) (2013)

[26] Hindle, A., Ernst, N.A., Godfrey, M.W., Mylopoulos, J.: Automated topic naming to support cross-project analysis of software maintenance activities. In: Proceedings of the International Working Conference on Mining Software Repositories (MSR), pp. 163–172 (2011)

[27] Howard, M.J., Gupta, S., Pollock, L., Vijay-Shanker, K.: Automatically mining software-based, semantically-similar words from comment-code mappings. In: Proceedings of the Working Conference on Mining Software Repositories (MSR) (2013)

[28] Kamiya, T., Kusumoto, S., Inoue, K.: CCFinder: A multilinguistic token-based code clone detection system for large scale source code. IEEE Transactions on Software Engineering **28**(7), 654–670 (2002)

[29] Kawrykow, D., Robillard, M.P.: Non-essential changes in version histories. In: Proceedings of the International Conference on Software Engineering, pp. 351–360 (2011)

[30] Koschke, R., Canfora, G., Czeranski, J.: Revisiting the delta ic approach to component recovery. Science of Computer Programming **60**(2), 171–188 (2006)

[31] Kuhn, H.W.: The hungarian method for the assignment problem. Naval Research Logistics Quarterly **2**, 83–97 (1955)

[32] Lawrie, D., Binkley, D.: Expanding identifiers to normalize source code vocabulary. In: Proceedings of the International Conference on Software Maintenance, (ICSM), pp. 113–122 (2011)

[33] Lawrie, D., Feild, H., Binkley, D.: Syntactic identifier conciseness and consistency. In: Proceedings of the International Workshop on Source Code Analysis and Manipulation, pp. 139–148 (2006)

[34] Lawrie, D., Morrell, C., Feild, H., Binkley, D.: What's in a name? a study of identifiers. In: Proceedings of the International Conference on Program Comprehension, pp. 3–12 (2006)

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

[36] Lehman, M.M.: Programs life cycles and laws of software evolution. Proceedings of the IEEE **68**(9), 1060–1076 (1980)

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

[38] Madani, N., Guerrouj, L., Di Penta, M., Guéhéneuc, Y.G., Antoniol, G.: Recognizing words from source code identifiers using speech recognition techniques. In: Proceedings of the European Conference on Software Maintenance and Reengineering, pp. 68–77 (2010)

[39] Maletic, J.I., Antoniol, G., Cleland-Huang, J., Hayes, J.H.: In: International Workshop on Traceability in Emerging Forms of Software Engineering, p. 462 (2005)

[40] Malpohl, G., Hunt, J.J., Tichy, W.F.: Renaming detection. In: Proceedings of the International Conference Automated Software Engineering, pp. 73–80 (2000)

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

[42] Marcus, M.P., Marcinkiewicz, M.A., Santorini, B.: Building a large annotated corpus of english: the penn treebank. Journal of Computational Linguistics - Special issue on using large corpora **19**(2), 313–330 (1993)

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

[44] Neamtiu, I., Foster, J.S., Hicks, M.: Understanding source code evolution using abstract syntax tree matching. Software Engineering Notes **30**(4), 1–5 (2005)

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

[46] Poshyvanyk, D., Marcus, A.: The conceptual coupling metrics for object-oriented systems. In: Proceedings of the International Conference on Software Maintenance, pp. 469–478 (2006)

[47] Prete, K., Rachatasumrit, N., Sudan, N., Kim, M.: Template-based reconstruction of complex refactorings. In: Proceedings of the International Conference on Software Maintenance (ICSM), pp. 1–10 (2010)

[48] Santayana, G.: The Life of Reason: Introduction and Reason in Common Sense, vol. 1. Charles Scribner's Sons (1905)

[49] Sheskin, D.J.: Handbook of Parametric and Nonparametric Statistical Procedures (fourth edition). Chapman & All (2007)

[50] Strauss, A.L.: Qualitative analysis for social scientists. Cambridge Univsersity Press (1987)

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

[52] Thummalapenta, S., Cerulo, L., Aversano, L., Di Penta, M.: An empirical study on the maintenance of source code clones. Empirical Software Engineering **15**(1), 1–34 (2010)

[53] Toutanova, K., Manning, C.D.: Enriching the knowledge sources used in a maximum entropy part-of-speech tagger. In: Proceedings of the Joint SIGDAT Conference on Empirical Methods in Natural Language Processing and Very Large Corpora (EMNLP/VLC-2000), pp. 63–70 (2000)

[54] Weissgerber, P., Diehl, S.: Identifying refactorings from source-code changes. In: Proceedings of the International Conference on Automated Software Engineering (ASE), pp. 231–240 (2006)

[55] Xing, Z., Stroulia, E.: Refactoring detection based on UMLDiff change-facts queries. In: Proceedings of Working Conference on Reverse Engineering, pp. 263–274 (2006)

[56] Yang, J., Tan, L.: SWordNet: Inferring semantically related words from software context. Empirical Software Engineering (2013)

[57] Zimmermann, T., Weisgerber, P.: Preprocessing CVS data for fine-grained analysis. In: Proceedings of the International Workshop on Mining Software Repositories, pp. 2–6 (2004)

[58] Zimmermann, T., Weissgerber, P., Diehl, S., Zeller, A.: Mining version histories to guide software changes. In: Proceedings of the International Conference on Software Engineering, pp. 563–572 (2004)

Venera Arnaoudova Venera Arnaoudova is a Ph.D. candidate at the École Polytechnique de Montréal under the supervision of Professor Giuliano Antoniol and Professor Yann-Gaël Guéhéneuc. In 2008, she received her master degree in Computer Science from Concordia University (Montréal, Canada) under the supervision of Professor Constantinos Constantinides. In 2006, she received her bachelor degree in Computer and Electrical Engineering (Major of Computer Science) from the engineering school PolytechLille (Lille, France).

Her research interest is in the domain of software evolution and particularly, analysis of source code lexicon and documentation, empirical software engineering, refactoring, patterns, and anti-patterns. Website: www.veneraarnaoudova.com

Laleh Eshkevari Laleh M. Eshkevari joined the PhD program of Department of Computer Science and Software Engineering of École Polytechnique de Montréal in fall 2009. She is a member of SOCCER laboratory and Ptit-Dej team directed by professor Giuliano Antoniol and Professor Yann-Gaël Guéhéneuc. She received her master degree in Computer Science in 2008 from Concordia University (Montréal, Canada) under the supervision of Professor Constantinos Constantinides and Professor Juergen Rilling. She received her bachelor degree in Mathematic Applied in Computer Science from Amirkabir University of Technology (Tehran, Iran). Her research interest is in the domain of software maintenance and evolution: change impact analysis, linguistic refactoring, source code analysis, programming languages, and empirical software engineering. Website: laleh-eshkevari.ca/

Massimiliano Di Penta Massimiliano Di Penta is associate professor at the University of Sannio, Italy. His research interests include software maintenance and evolution, reverse engineering, empirical software engineering, search-based software engineering, and service-centric software engineering. He is author of over 180 papers appeared in international conferences and journals. He serves and has served in the organizing and program committees of over 100 conferences such as ICSE, FSE, ASE, ICSM, ICPC, CSMR, GECCO, MSR, SCAM, WCRE, and others. He has been general chair of SCAM 2010, WSE 2008, general co-chair of SSBSE 2010, WCRE 2008, and program co-chair of MSR 2013 and 2012, ICPC 2013, ICSM 2012, SSBSE 2009, WCRE 2006 and 2007, IWPSE 2007, WSE 2007, SCAM 2006, STEP 2005, and of other workshops. He is steering committee member of ICSM, CSMR, IWPSE, SSBSE, PROMISE, and past steering committee member of ICPC, SCAM, and WCRE. He is in the editorial board of He is in the editorial board of IEEE Transactions on Software Engineering, the Empirical Software Engineering Journal edited by Springer, and of the Journal of Software: Evolution and Processes edited by Wiley. He is member of IEEE, IEEE Computer Society, and of the ACM. Further info on www.rcost.unisannio.it/mdipenta

Rocco Oliveto Rocco Oliveto is Assistant Professor in the Department of Bioscience and Territory at University of Molise (Italy). He is the Director of the Laboratory of Informatics and Computational Science of the University of Molise. He received the PhD in Computer Science from University of Salerno (Italy) in 2008. His research interests include traceability management, information retrieval, software maintenance and evolution, search-based software engineering, and empirical software engineering. He serves and has served as organizing and program committee member of international conferences in the field of software engineering. In particular, he was the program co-chair of TEFSE 2009, the Traceability Challenge Chair of TEFSE 2011, the Industrial Track Chair of WCRE 2011, the Tool Demo Co-chair of ICSM 2011, the program co-chair of WCRE 2012, and he will be the program co-chair of WCRE 2013, SCAM 2014, and ICPC 2015. He is member of IEEE Computer Society, ACM, and IEEE-CS Awards and Recognition Committee.

Giuliano Antoniol Giuliano Antoniol (Giulio) Canada Research Chair in Software Change and Evolution, served as program chair, industrial chair, tutorial, and general chair of international conferences and workshops. He contributed to the program committees of more than 30 IEEE and ACM conferences and workshops. He is a member of the editorial boards of five software engineering journals and he acts as referee for all major software engineering journals.

He is currently Full Professor at the Polytechnique Montréal, where he works in the area of software evolution, empirical software engineering, software traceability, search based software engineering, software testing and software maintenance.

Yann-Gaël Guéhéneuc Yann-Gaël Guéhéneuc is full professor at the Department of computer and software engineering of École Polytechnique de Montréal where he leads the Ptidej team on evaluating and enhancing the quality of object-oriented programs by promoting the use of patterns, at the language-, design-, or architectural-levels. He is IEEE Senior Member since 2010. In 2009, he was awarded the NSERC Research Chair Tier II on Software Patterns and Patterns of Software. He holds a Ph.D. in software engineering from University of Nantes, France (under Professor Pierre Cointe's supervision) since 2003 and an Engineering Diploma from École des Mines of Nantes since 1998. His Ph.D. thesis was funded by Object Technology International, Inc. (now IBM OTI Labs.), where he worked in 1999 and 2000. His research interests are program understanding and program quality during development and maintenance, in particular through the use and the identification of recurring patterns. He was the first to use explanation-based constraint programming in the context of software engineering to identify occurrences of patterns. He is interested also in empirical software engineering; he uses eye-trackers to understand and to develop theories about program comprehension. He has published many papers in international conferences and journals, including IEEE TSE, Springer EMSE, ACM/IEEE ICSE, and IEEE ICSM. He is currently (2013-2014) in sabbatical in Korea, working with colleagues at KAIST, Yonsei U., and SNU.