2011 11th IEEE International Working Conference on Source Code Analysis and Manipulation

The Effect of Lexicon Bad Smells on Concept Location in Source Code

Surafel Lemma Abebe[1], Sonia Haiduc[2], Paolo Tonella[1], Andrian Marcus[2]

[1]Software Engineering Research Unit
Fondazione Bruno Kessler
Trento, Italy

[2]Department of Computer Science
Wayne State University
Detroit, MI, USA

surafel@fbk.eu; sonja@wayne.edu; tonella@fbk.eu; amarcus@wayne.edu

***Abstract*— Experienced programmers choose identifier names carefully, in the attempt to convey information about the role and behavior of the labeled code entity in a concise and expressive way. In fact, during program understanding the names given to code entities represent one of the major sources of information used by developers. We conjecture that lexicon bad smells, such as, extreme contractions, inconsistent term use, odd grammatical structure, etc., can hinder the execution of maintenance tasks which rely on program understanding. We propose an approach to determine the extent of this impact and instantiate it on the task of concept location. In particular, we conducted a study on two open source software systems where we investigated how lexicon bad smells affect Information Retrieval-based concept location. In this study, the classes changed in response to past modification requests are located before and after lexicon bad smells are identified and removed from the source code. The results indicate that lexicon bad smells impact concept location when using IR-based techniques.**

Keywords-lexicon bad smells; program comprehension; concept location; text retrieval; software lexicon; code smells.

I. INTRODUCTION

While performing maintenance activities, developers spend a considerable amount of time reading and understanding the source code of the software system they are working on. During this task, identifiers and comments play an important role, as they connect domain concepts to their representation in the software and often capture the software intent. Flaws in the naming of identifiers could affect program comprehension and, in consequence, the associated maintenance tasks (i.e., concept location, impact analysis, change propagation, etc.). We call such flaws *lexicon bad smells* in software [1] and they can usually be fixed through renaming. Examples of bad smells include using extremely contracted terms in the names of identifiers (e.g., "dg" instead of "diagonal"), misspelled terms (e.g., "docment" instead of "document"), etc. We consider a high-quality identifier one having few to none lexicon bad smells, and a low quality identifier one that has numerous such smells. Identifiers are composed from one or more words, and each of them can exhibit distinct bad smells.

The need for high quality identifiers in source code has been underlined on numerous occasions [2-8]. Recently, the quality of identifiers has been also linked to other quality properties of the software, like the number of detected bugs [9-12], cyclomatic complexity [11], and code readability [11]. In spite of these studies, the effect, if any, of the identifier quality on daily comprehension tasks performed by developers on source code is still unknown. Questions like "Do low quality identifiers make program comprehension, and thus maintenance tasks more difficult to perform?" remain unanswered. Our long term goal is answering such research questions.

In this paper, we investigate the effect of lexicon bad smells on one of the program comprehension tasks, concept location, using *reenactment in before-and-after* studies. In order to carry out the investigation, we performed a case study on two open source object-oriented software systems. We performed concept location using two different Information Retrieval (IR) techniques before and after the bad smells are identified and removed from the code and compared the outcomes. The results indicate that lexicon bad smell removal had a significant positive impact on the IR-based concept location in one of the systems, whereas the impact was not statistically significant, even though it was in the positive direction, for the second system. The results also revealed that the IR technique of choice is not an influencing factor, even though some IR techniques may benefit from lexicon improvement more than others.

The rest of the paper is organized as follows: Section II gives background information on IR-based concept location; Section III presents the approach for determining the impact of lexicon bad smells on comprehension activities; Section IV describes the case study we performed; Section V reviews the related work; and Section VI concludes the paper and describes future directions.

II. IR-BASED CONCEPT LOCATION IN SOURCE CODE

Concept location is a critical activity during software evolution as it produces the location where a change is to start in response to a change request, such as, a bug report or a new feature request. Concept location is one of the most common and critical comprehension activities undertaken during software change [18]. The software change process starts with a modification request and ends with a set of changes to the existing code and addition of new code. The software maintainer undertakes a set of activities to

978-0-7695-4347-5/11 $26.00 © 2011 IEEE
DOI 10.1109/SCAM.2011.18

determine the parts of the software that need to be changed: concept location, impact analysis, change propagation, and sometimes refactoring. Concept location starts with the change request and succeeds when the developer finds the location in the source code where the first change must be made (e.g., a class).

Various approaches have been proposed for performing concept location, including static, dynamic and mixed techniques. One of the most popular static approaches is using a form of textual search in order to search the source code of the system for relevant classes, using a query formulated by the developer or extracted from the change request. Marcus et al. [14] proposed an Information Retrieval (IR) -based approach to concept location. The idea of the approach is to treat source code as a text corpus and use IR methods to index the corpus and build a search engine, which allows developers to search the source code much like they search other sources of digital information (e.g., the internet). The methodology was further extended to make use of other sources of information [15-16]. Here we use the original IR-based approach [14].

IR-based concept location is composed of five major steps. These steps may be instantiated differently based on the type of IR method used and how the corpus is created.

1. *Corpus creation.* In this step, the version of the source code where the changes need to be implemented is parsed using a developer-defined granularity level (classes, methods, etc.). A corpus is constructed such that each code entity found at the chosen granularity level will have a corresponding document in the corpus. Natural language processing (NLP) techniques and filtering techniques can be applied to the corpus. Usually, the identifiers are split according to common naming conventions. For example, "setValue", "set_value", "SETValue", etc. are all split to "set" and "value". The original identifiers can be also kept in the corpus, in order to account for any identifiers that might be included in a query. Filtering is also used to eliminate programming language specific keywords (Java and C++), and also common English stop words[1].
2. *Indexing*. The corpus created in the previous step is indexed using the chosen IR method and a mathematical representation of the corpus is created. Each document in the corpus has a corresponding index.
3. *Query formulation*. IR-based concept location uses a query in order to search the source code of a system for relevant documents. This query can be formulated by the developer by analyzing the change request and selecting terms which she considers to be relevant, by gathering a set of relevant terms from the system documentation, etc. The simplest approach is to use the whole change request as the query. The query is then processed in the same way as the corpus, i.e., by splitting any identifiers present in the change request and by filtering out programming keywords and common English words.
4. *Ranking documents*. In this step, similarities between the query and every document in the index are computed and the documents are ranked according to this similarity. The similarity measure depends on the IR method used. In the case of a vector-based representation of the documents, the cosine similarity between the vectors is the most widely used measure.
5. *Results examination*. When a developer is involved in the concept location process, in this step she examines the ranked list of source code documents, starting with those with the highest similarities. For every source code document examined, a decision is required whether the document will be changed or not. If it will be changed, then the search succeeded and concept location ends. Else, if new knowledge obtained from the investigated documents helps formulate a better query (e.g., narrow down the search criteria), then step 3 should be reapplied. Otherwise, the next document in the list should be examined. This last step is optional, and it is often not performed in the case of studies where developers are not involved.

[1] http://www.webconfs.com/stop-words.php

III. APPROACH

In order to study the impact that lexicon bad smells have on program comprehension activities, we propose an approach based on *before-and-after studies.* These studies originate in biomedical sciences, where they have often been used to study the impact of an intervention on a particular condition present in patients [13]. Making an analogy, in our case the *"patients"* are the software systems developers need to understand in order to carry out a specific maintenance task, the *"condition"* they are suffering from is the presence of lexicon bad smells, and the *"intervention"* performed is the removal of lexicon bad smells. The variable we want to measure is the effort involved in understanding and the quality of the outcome.

The main advantage of before-and-after studies is the fact that they do not require a patient control group. Thus, instead of comparing subjects who suffered an intervention to those who did not, in before-and-after studies the comparison is made on the same subjects, before and after the intervention. As we want to study the impact of lexicon bad smells on software maintenance tasks performed by developers on the same software systems, before-and-after studies are particularly suited for our purposes.

Having developers performing the maintenance tasks on software systems before and after removing lexicon bad smells, however, has several limitations which make it very difficult to perform in practice. First, it requires a lot of time and effort and finding developers willing to invest the time needed to perform the changes can be very difficult. Second, it also introduces an additional variable, i.e., the developer, which is very difficult to control, and can, thus, introduce bias in the study.

An alternative we use here, which has been applied well in previous comprehension studies [15-16], is based on *reenactment* [17]. We use historical data to determine the outcome of a task that was performed in the past. For example, using bug tracking systems and versioning systems,

we can find out what parts of the code were changed in response to a bug fix request. In the context of concept location, we call these *target changes* (target classes, target methods, etc.), as they are the targets of the concept location process. The details are missing (i.e., we do not know exactly how the changes were determined) but we know what the input and output is. During reenactment the task is carried by a user or simulated by a tool and the goal is to obtain the same output starting from the given input. When the expected output is obtained, we consider that the task is performed correctly.

The main advantage of reenactment is that it allows for automation, i.e., the developer actions can be simulated by tools, thus allowing the collection of far more data points. A positive side effect is eliminating the user-induced bias. Reenactment based studies have associated costs and limitations, though. They require the existence of data on prior task executions. For example, in the case of concept location, we need to correlate past changes with bug or feature descriptions. In addition, when tools are used to simulate the developer actions, the analysis is usually focused on the best, worst, and/or average cases. These do not always correspond to actual cases that occur in real life settings, when humans are involved. We believe that the positive outweighs the negative in this case, and thus perform automatic reenactment in our study. Ideally, both kinds of studies should be employed, whenever possible.

In this work, we use reenactment in a before-and-after study in order to analyze the impact lexicon bad smells have on the effort of software comprehension activities. To automate the reenactment process we have used concept location tools that implement Information Retrieval (IR) techniques.

In studies that use IR-based concept location, it is common to use the ranks of the target classes as effort measures [15-16]. These represent the number of classes a developer would have to look at before finding the target classes if she would analyze the list of results in the order provided. Clearly this is an approximate measure as it considers that the effort to investigate a class is identical for all classes, which may not be the case in real-life scenarios. However, the measure is consistently applied in each treatment. Finding one of the target classes implies correctness of concept location. The impact of the lexicon bad smells on IR-based concept location can be, therefore, measured using the difference between the ranks of the target classes before and after the refactoring. If these ranks improve (i.e., less effort is required to locate the classes) then we can conclude that removing the lexicon bad smells improves IR-based concept location.

A. Lexicon bad smells detection

The first step in the proposed approach deals with the detection of lexicon bad smells. Since manual inspection of the source code to identify lexicon bad smells is a very tedious, difficult, and error prone task, this step needs to be automated. We have previously implemented and evaluated the precision of a suite of tools, called *LBSDetectors* [1], which automatically locate and report lexicon bad smells in source code. The settings of the *LBSDetectors* can be specified using customizable configuration files and thresholds, which makes them adaptable to specific needs in a particular software system.

The tools are currently able to determine smells in class, method, and attribute identifiers. The set of bad smells they are capable of detecting is described below. However, this set of bad smells is not exhaustive and other lexicon bad smells and bad smell detectors can be defined.

Extreme contraction: Extremely short terms, due to an excessive word contraction, abbreviation, or acronym are used in an identifier.
Example: itemSel (Sel=Selected), aSz (a=array, sz=size)
Exceptions: This rule does not apply to:

- Extreme contractions included in the naming conventions adopted in the system (e.g., `m_` is a prefix used in the Hungarian notation to mark attributes of a class).
- Common programming and domain terms (e.g., `msg`, `SQL`, etc.).
- Short dictionary words (e.g., `on`, `it`, etc.).

Refactoring: Rename identifiers using longer, more expressive terms.

Inconsistent identifier use: A concept is not represented by identifiers in a consistent and concise way. An identifier is considered inconsistent, when it is contained in another identifier of the same type (e.g., class/method/attribute name), which is found in the same container entity (e.g., package, class).
Example:

```
class Documents {
  private String absolute_path;
  private String relative_path;
  private String path;//path is inconsistent
}
```

Refactoring: Rename identifiers to make them concise and consistent.

Meaningless terms: Meaningless metasyntactic variables are used in an identifier.
Example: foo, bar
Refactoring: Rename identifiers using meaningful terms.

Misspelling: At least one of the words (excluding accepted abbreviations, contractions, and acronyms) used to construct an identifier are misspelled.
Example: conector (instead of connector)
Refactoring. Correct the misspelled words.

Odd grammatical structure: The grammatical structure of an identifier is not appropriate for the specific kind of software entity it represents (e.g., a class name contains a verb, method names do not start with a verb, etc.).
Example:

```
class Compute //compute is a verb {
  public void addition();//addition is a noun
}
```

*Refactoring***:** The identifier should be renamed following the proper syntactic rules for the specific entity it represents.

Overloaded identifiers: An identifier is overloaded with multiple semantics, which might indicate the entity it represents is also overloaded with multiple responsibilities (e.g., a method name contains two verbs).
Example:
```
//Two tasks: creating and exporting a list
create_export_list();
```
Refactoring: Split the entity into two (or more) entities, each having a single responsibility and name each entity using an appropriate (non-overloaded) identifier.

Useless type indication: The type of a variable is explicitly indicated in its identifier, making it redundant.
Example:
```
class Rental {
  short key_short; // type in attribute name
}
```
Exceptions: This rule does not apply in the following cases:
- A static attribute used to realize the singleton design pattern, which usually has the same identifier as the class.
- Individual characters or groups of characters used to denote the type of the variable, which are included in adopted naming conventions (i.e., in the Hungarian notation, *i* is a prefix used in identifiers of integer value).

Refactoring: Rename the identifier by removing the type name and, if necessary, rename it such that it conveys information about its role in the program.

Whole-part: A term is used to name a concept and appears also in the name of its properties or operations.
Example:
```
class Account {
  int account; //Ambiguous use
  void computeAccount();
  //Account is redundant information
}
```
Exceptions: A static attribute, used to realize the singleton design pattern has usually the same identifier as the class. Constructor methods have the same name as the class.
Refactoring: Rename different entities so as to differentiate their role and/or avoid redundant information.

Synonyms and similar identifiers. Synonyms or similar terms are used to construct the identifiers representing different entities declared in the same container, such that differentiating between their responsibilities is difficult.
Example:
```
class User {
  public String idCopy(String text);
  // Replica is synonym to Copy - confusing
  public String idReplica(String text); }
```
Refactoring. Rename the different entities so as to differentiate their role/functionality. If necessary, introduce a common superclass or interface for the shared properties.

Terms in wrong context. Using terms that pertain to the domain of another container (e.g., package) indicate that the entity named by such terms may be misplaced.
Example.
```
package collections;
class IntArray;
// TypeDetector is in the wrong
// package or is incorrectly named
class TypeDetector ;
package detectors;
class MuonDetector ;
class PhosDetector ;
class HLTDetector ;
```
Refactoring: Move the misplaced entity to the container it logically belongs to or rename it to better reflect the role it has in its currently assigned container.

No hyponymy/hypernymy in class hierarchies. The identifier of a child class in an inheritance hierarchy is not a hyponym of the identifier of its parent class.
Example:
```
class Mammal ;
  // Violin is not a hyponym of mammal
  class Violin extends Mammal ;
```
Exception: When class identifiers are compound words or they contain abbreviations, contractions, or acronyms, hyponymy and hypernymy can be hard to assess.
Refactoring: Refactoring may just require identifier renaming, or may involve deeper restructuring of the inheritance hierarchy.

Identifier construction rules: The name of an identifier does not follow a standard naming convention adopted in the system, which makes use of prefixes, suffixes and/or term separators.
Example.
```
class StudentInformation {
  private String fName;
  // should start with an f
  private String Address;
}
```
Refactoring: Restructure the identifier by following the adopted naming convention.

B. Refactoring the lexicon bad smells

The next step in the process is refactoring the lexicon bad smells detected. For now, the bad smells are refactored manually. Part of this step could be eventually automated, but refactoring some of the bad smells might still require the intervention of a human, in order to ensure correctness and meaningfulness.

The refactoring is done using clues found in the source code comments and programming constructs, as well as in the documentation. Also, developers use at this time any previous knowledge about the system and its domain they might have. The renaming is made according to the rules described in the previous section.

C. Reenactment in the context of before and after studies

The last part of the process involves three main activities. First, we carry out IR-based concept location on the original source code (i.e., containing the lexicon bad smells) and record the effort (i.e., the rank of the target classes) for performing the task, as well as its correctness. This represents the baseline data, *before* the intervention, i.e., before fixing the lexicon bad smells. Next, the same IR-based concept location tasks are performed on the refactored source code, where the lexicon bad smells have been removed. The effort (i.e., the ranks of the target classes)

needed to perform the task and the correctness are again recorded and this represents the data *after* the intervention. At the end, the two effort measurements (i.e., *before* and *after* the refactoring) are compared and the difference represents the impact that lexicon bad smells have on the task investigated.

IV. CASE STUDY

In this section, we describe a case study carried out following the above approach to investigate the impact of lexicon bad smells on concept location in two open source software systems.

A. Systems, bugs, queries, and IR techniques

The two software systems used in the study are FileZilla Client[2] 3.0.0, an open source, cross-platform, graphical FTP, FTPS, and SFTP client and OpenOffice[3] 1.0.0, a well-known open source office software suite for word processing, spreadsheets, presentations, databases, etc. FileZilla is medium-sized, having 209 classes, while OpenOffice is a large system, with almost 12,000 classes. Both are implemented mostly in C++.

The corpus of the two systems was extracted both before and after the lexicon bad smells were fixed, and identifiers were split, keeping the originals. Also, filtering was applied in order to remove common English terms and C++ keywords.

Both systems had online bug tracking systems, from which we extracted a set of 29 bugs for FileZilla and 19 for OpenOffice. The bugs were selected such that the target classes could be identified, either from the patches that sometimes accompany the bug reports in the bug tracking system, or from commits to the versioning system hosting the source code. In this second case, the target classes were located by identifying the bug ID in the commit messages found in the source code versioning system and then analyzing the changes that occurred in those commits. The bug reports were filtered such that the commits included only one bug ID in the commit message.

We used the concatenation of the title and the description of the bugs, as retrieved from the bug reports, as the queries for IR-based concept location. This approach is well suited for reenactment studies on concept location techniques, as the developer variable is removed, and it has been previously used successfully for this purpose [15]. We applied the same processing as for the corpus (i.e., splitting and filtering) on the queries.

In our study, we used two different IR techniques for indexing and searching the software for concept location: Latent Semantic Indexing (LSI) [14], and an improved version of the Vector Space Model implemented in Lucene[4]. We used these two different IR techniques in order to observe if the effect of the lexicon bad smells on concept location depends on the IR engine used or not.

LSI takes different settings than Lucene, like the number of dimensions to which the vector space should be reduced during the Singular Value Decomposition (SVD) and the weight to be used when scoring documents. In this study, we used 100 dimensions for SVD and tf-idf as the weight. We make a note that the focus of this study was to observe the difference in performance when lexicon bad smells are present versus when they are absent, given that all the other variables are set, including those used for LSI. Thus, it is of less importance which dimension or weight is used, as long as they are the same before and after refactoring the bad smells.

We performed all the five steps described in Section II for both the original source code of the systems and the refactored one. We used class level granularity.

B. Bad smells detection and refactoring

We focused on eight of the twelve types of bad smells described in Section II, for which we could get enough information from the artifacts of the systems to detect the smells and suggest new names. For the other four smell types, we did not have enough information about the systems in order to provide a reliable renaming of the identifiers. For example, in the case of the *synonyms and similar identifiers* bad smell, in order to provide good names we would need a glossary of the system where all the synonyms used in the source code to refer to a particular concept would be listed. Such a glossary does not currently exist for either of the two systems. Note that using a general-purpose dictionary like WordNet would not suffice, as many times the word relationships in source code are not the same as in natural language [19].

The lexicon bad smells we focused on in this study are *extreme contraction, inconsistent identifier use, meaningless terms, misspelling, odd grammatical structure, overloaded identifiers, useless type indication,* and *whole part*.

For the bad smell detection, we used the *LBSDetectors* on the target classes for the set of bugs selected in the two systems. We found 192 identifiers containing at least one bad smell in the 28 unique target classes in FileZilla 3.0.0 and 775 identifiers for the 26 unique target classes in OpenOffice 1.0.0 (see Table I). Among the eight lexicon bad smells, the number of identifiers detected to contain *odd grammatical structure, extreme contraction, misspelling and inconsistent identifiers* was high, while the number of *meaningless terms* was low in both systems (see Table I).

As mentioned before, the bad smell correction was done manually for each identifier. For example, the method name *command* was identified as an odd grammatical structure bad smell because it does not contain a verb. It was renamed to *executeCommand* after consulting the comment of the method in the source code of *FileZilla*. Not all identified bad smells were corrected, though. For example, the extreme contractions which were due to the use of the Hungarian notation were not changed, but other extreme contractions, such as, *Lev* (refactored to *Levenshtein*) and *Exc* (refactored to *Excel*), were refactored. *Inconsistent identifiers,* detected in large numbers in FileZilla, referred generally to method identifiers which were included within other method names

[2] http://filezilla-project.org/

[3] http://www.openoffice.org/

[4] http://lucene.apache.org/

in the same class, thus making their meaning not specific enough. An example of such a method name was *Connect,* which appeared also in the name of another method, *ConnectToClient* inside the same class in FileZilla. It is thus unclear how the two methods are different and to what exactly *Connect* refers to. In consequence, based on the body of the method and the comments, we renamed the identifier to *ConnectToServer,* which is more specific and reflects the functionality of the method better. For the *misspellings* bad smell, a high number of bad smelling identifiers were reported in OpenOffice. One example of such an identifier was *isApplyable,* which was renamed to *isApplicable*. There were almost no identifiers containing *meaningless terms* in either system, as the terms in our predefined list of metasyntactic variable names did not occur in the analyzed systems. The only exception was *var*, which occurred in one target class.

TABLE I. NUMBER OF IDENTIFIERS CONTAINING BAD SMELLS IN THE TARGET CLASSES AND NUMBER OF REFACTORED IDENTIFIER OCCURENCES IN FILEZILLA 3.0.0 AND OPENOFFICE 1.0.0

Bad Smell	FileZilla	OpenOffice
Extreme contraction	86	480
Inconsistent identifier use	95	74
Meaningless terms	0	1
Misspelling	64	436
Odd grammatical structure	147	434
Overloaded identifiers	4	12
Useless type indication	2	7
Whole-part	13	25
Number of identifiers containing bad smells in target classes	192	775
Number of identifier occurrences refactored in the system	2,216	90,749
Number of unique target classes	28	26

Two of the authors independently proposed names for refactoring the lexicon bad smells detected and then compared their suggestions. In the cases where the names proposed by the two authors were different, there was a discussion about the two names and an agreement was reached. While suggesting the new names the two authors performed the actions listed in Table II. The most frequent action was *term expansion*, where extremely contracted terms were expanded to the terms they were referring to (e.g., *nTrot* expanded to *nTextRotation*). Other frequent actions were *addition* and *deletion*. Addition included adding missing verbs to method names or replacing a term with a meaningful one. In OpenOffice, we also encountered a few identifiers that contained German terms, which we replaced with their English translation (example: *importGraf* renamed to *importGraphic*).

All renamings were performed across the entire system. At the same time, if the bad smells occurred in the bug titles and descriptions, they were renamed also there. The total number of occurrences of bad smelling identifiers changed in the whole system was 2,216 for FileZilla 3.0.0 and 90,749 for OpenOffice 1.0.0 (see Table I).

TABLE II. TYPES OF ACTIONS PERFORMED TO FIX THE LEXICON BAD SMELLS AND THE CORRESPONDING NUMBER OF IDENTIFIERS ON WHICH THEY ARE APPLIED

Type of action while correcting a smell	OpenOffice	FileZilla
Term expansion	484	38
Spelling correction	2	0
Term reordering	35	31
Added term	283	71
Deleted term	139	42
Replaced term	138	37
Language translation	33	0

C. Results and discussion

For each bug in the two systems we reenacted concept location using the title and description of the bug together as the query. We simulated the user by using this initial query and no subsequent query reformulations. We assumed the users would inspect the classes in the order suggested by the tool. For each bug we performed four reenactments: two before the bad smells removal, using LSI and Lucene, respectively, and two after. In each run we recorded the effort measures, represented by the rank of the target classes in the list of search results. The measures for FileZilla and OpenOffice are reported in Table III and Table IV, respectively. We present the results for each of the two systems separately and then discuss the differences between them.

For FileZilla, when using Lucene, out of the 45 non-unique target classes for the 29 bug reports selected, 21 had the same rank in the list of search results before and after the refactoring. At the same time, the results for 13 target classes were worse after refactoring. For the remaining 11 classes the results were better after refactoring the lexicon bad smells. Although there were more target classes for which the ranking did not improve, the overall ranking of the target classes slightly improved after the bad smells were removed (see Table V).

The absolute difference between the ranks of the target classes before and after refactoring, which is the sum of all the individual differences for each target class, is 14, and the average difference is 0.31. This indicates an overall improvement of 14 positions in the list of ranked results over all target classes after the lexicon bad smells were removed, with an average improvement of 0.31 positions for each target class. The distribution of the deltas can be seen in the histogram presented in Figure 1a. The overall improvement is due to the fact that a few classes had a significant improvement in the rank, which overcame the decrease in other target classes (see Figure 1a). In fact, the average (4.27) and maximum (29) positive deltas were higher than the average (-2.53) and minimum (-12) negative deltas.

In order to see if the difference between results before and after the refactoring is statistically significant, we performed a two-tailed paired t-test between the two series of data. The p-value of 0.95 indicates that there is no statistical proof that the refactoring had an effect on the ranks of the target classes.

TABLE III. THE RANK OF CHANGED CLASSES IN THE LIST OF SEARCH RESULTS WHEN USING LSI AND LUCENE ON THE ORIGINAL AND REFACTORED FILEZILLA SOURCE CODE

No.	Bug ID	LSI		Lucene	
		Before	*After*	*Before*	*After*
1	1299	68, 54	65, 56	24, 17	26, 19
2	3023	36	37	20	19
3	3198	51, 2, 84	53, 3, 86	4, 1, 2	5, 1, 2
4	3220	68, 39, 17, 45	67, 40, 17, 45	65, 16, 11, 1	63, 16, 11, 1
5	3230	33	32	1	1
6	3232	2	1	4	2
7	3235	178, 52, 13	91, 50, 10,	130, 85, 64	101, 86, 65
8	3239	28	50	1	1
9	3252	6	9	8	6
10	3270	52	52	2	2
11	3272	84	81	16	17
12	3278	72	80	2	3
13	3284	91	86	8	8
14	3287	51	52	2	2
15	3307	68	67	2	2
16	3308	21	21	4	2
17	3319	124	122	25	24
18	3323	3	3	2	2
19	3334	64	62	7	6
20	3341	66	61	7	7
21	3343	21	19	4	4
22	3344	57	54	5	5
23	3345	3	5	4	6
24	3348	26	52	3	5
25	3356	53, 34, 47, 25, 68, 67, 116	51, 45, 41, 25, 68, 66, 117	11, 27, 22, 24, 1, 10, 12	12, 33, 23, 20, 1, 9, 10
26	3372	24	50	1	1
27	3373	51, 101	55, 99	1, 55	1, 67
28	3397	17, 25	17, 25	2, 3	2, 3
29	3403	42	67	1	1

When using LSI for FileZilla, there were 9 cases in which the ranking of the target classes was the same before and after refactoring, 17 cases in which the ranking was worse and 19 where the results were better after refactoring. However, the absolute delta between ranks before and after refactoring was -6, indicating a slight decrease in the results (-0.13 positions per target class). This time, even though the maximum delta value was an improvement of 87 positions (see Figure 1b), the average rank decrease (-8.12) was higher than the average improvement (6.95). When performing the paired two-tailed t-test between the results before and after the refactoring, the p-value was 0.68, indicating again that there is little statistical significance for the impact of the refactoring on the ranks of the target classes.

The results for FileZilla indicate that removing the lexicon bad smells has little impact overall, when considering the effect on all target classes. However, the effect on the ranking was significant in the case of some target classes, which registered an improvement in rank of almost 50% (first target class in bug 3235). At the same time, the study suggests that the IR technique used might have a small impact on the overall difference in ranks before and after refactoring for FileZilla. The difference between absolute deltas for Lucene and LSI was 20, with an average of 0.44 delta per target class. Although the delta in ranks before and after refactoring was not greatly affected by the IR technique used, Lucene generally placed the target classes higher than LSI in the list of search results, by an average of 34 ranks both before and after refactoring.

TABLE IV. THE RANK OF CHANGED CLASSES IN THE LIST OF SEARCH RESULTS WHEN USING LSI AND LUCENE ON THE ORIGINAL AND REFACTORED OPENOFFICE SOURCE CODE

No.	Bug ID	LSI		Lucene	
		Before	*After*	*Before*	*After*
1	4378	691	453	1174	29
2	5923	49	51	48	47
3	6906	1894	1671	48	44
4	7114	7366	7932	304	5
5	7868	6540	6919	95	79
6	8148	3531	3669	177	44
7	8426	5222, 2814, 2613	4039, 2169, 1349	2, 21, 57	1, 11, 4
8	8640	126	79	29	12
9	8755	3222	3489	434	434
10	8779	120	142	1146	1220
11	9391	431, 7102	639, 10749	2, 3	2, 5
12	9959	2185, 3132	1347, 3300	32, 8	22, 12
13	10424	1380	1141	780	798
14	10532	7199, 1621	6323, 1165	1757, 535	1766, 278
15	10828	915	747	1	1
16	10995	4560, 1152, 40	4029, 1193, 50	444, 277, 2298	57, 11, 126
17	11776	7279, 4357	6346, 3911	82, 18	10, 13
18	17620	2023, 9093, 9292, 4112, 10058, 2099, 5250	1181, 3922, 8533, 4792, 10137, 1999, 5541	1, 609, 8260, 2, 987, 14, 790	1, 195, 6599, 2, 970, 11, 534
19	101603	583	729	274	85

For OpenOffice, the results were very different than those obtained for FileZilla. The results after refactoring were significantly better than the results before the refactoring was performed, for both IR techniques.

For Lucene, there were 23 cases out of the 33 non-unique target classes for which the results after refactoring were better than before, 5 classes for which the results were better before refactoring, and 5 classes which had no change in rank (see Table IV). The distribution of the deltas for all 33 non-unique target classes for the 19 bug reports can be found in Figure 1c. The absolute delta between the ranks of the target classes before and after refactoring was 7,281 (see Table V), with an average delta of 221. This means that after refactoring, the target classes were ranked 221 positions higher in the list of search results, on average. As for FileZilla, we computed the t-test between the series of target class ranks before and after the lexicon bad smell refactoring was performed, in order to see if the difference between the two data series was statistically significant. The p-value obtained was 0.02, indicating that refactoring had a positive effect on the results and this effect was statistically significant in OpenOffice, when using Lucene as the IR technique.

When using LSI on OpenOffice, there were 18 classes for which refactoring brought an improvement in their rank and 15 for which the results were worse after the refactoring.

TABLE V. SUMMARY STATISTICS FOR THE RANK DELTA IN FILEZILLA 3.0.0 AND OPENOFFICE 1.0.0. A POSITIVE DELTA INDICATES IMPROVED RANK AFTER BAD SMELL FIXING.

Statistics	FileZilla 3.0.0		OpenOffice 1.0.0	
	LSI	Lucene	LSI	Lucene
Absolute rank delta	-6	+14	8,315	7,281
Average delta (std dev)	-0.13 (15.4)	+0.31 (4.9)	251.97 (1212.9)	220.64 (495.7)
Average positive delta	6.95	4.27	831.06	321.22
Maximum positive delta	87	29	5171	2172
Average negative delta	-8.12	-2.53	-443.93	-21.4
Minimum negative delta	-26	-12	-3,647	-74
Median delta	0	0	100	10
Delta t-test p-value	0.95	0.68	0.24	0.02

The absolute delta was in this case 8,315, with an average of difference in ranks of 252 per target class. Thus, even when using LSI, refactoring the lexicon bad smells significantly improves the rank of the target classes in the list of search results, with a maximum improvement of 5,171 positions, for the second target class of bug 17620 (see Table IV).

In this case, the p-value obtained for the t-test between the target class ranks before and after the refactoring was 0.24. This is not statistically significant according to the 5% rule; however, positive average of the delta ranks (252) indicates that there was a positive effect on the ranks of the target classes when refactoring the lexicon bad smells in OpenOffice and using LSI for concept location.

One example of a significant improvement in the ranking of the target classes after the refactoring of lexicon bad smells is in the case of the first bug for OpenOffice, i.e., bug 4378. The only target class for this bug, i.e., *ExcXf8,* was initially located on position 1,174 in the list of results obtained when searching the source code of the system using Lucene and on position 691 when using LSI. The class originally contained 10 bad smelling identifiers, having a total of 22 lexicon bad smells. These were spread among three categories: extreme contraction (13 bad smells), misspelled terms (5 bad smells), and odd grammatical structure (4 bad smells). After refactoring, the bad smells were removed, which resulted in a significant improvement in the rank of the target class, i.e., position 29 with Lucene (improvement of 1,145 positions) and 453 with LSI (improvement of 242 positions). The improvement in rank can be attributed to the meaningful terms introduced in the identifiers and thus in the corpus after expanding the abbreviations and acronyms (e.g., *ExcXf8* was expanded to *ExcelFile8*, *nTrot* expanded to *nTextRotation*). This increased the number of common terms between the bug description and corpus (e.g., the term *excel* appeared in the corpus only after refactoring) and also increased the frequency of other common terms (e.g., the frequency of the term *rotation* has changed from 1 in the original corpus to 6 in the refactored corpus). Table VI shows the description of *Bug 4378*, the original identifiers with lexicon bad smells, the same identifiers after refactoring, and the terms contained only in the original and then refactored class.

TABLE VI. EXAMPLE OF REFACTORING THAT LED TO A SIGNIFICANT IMPROVEMENT IN RANK FOR THE TARGET CLASS

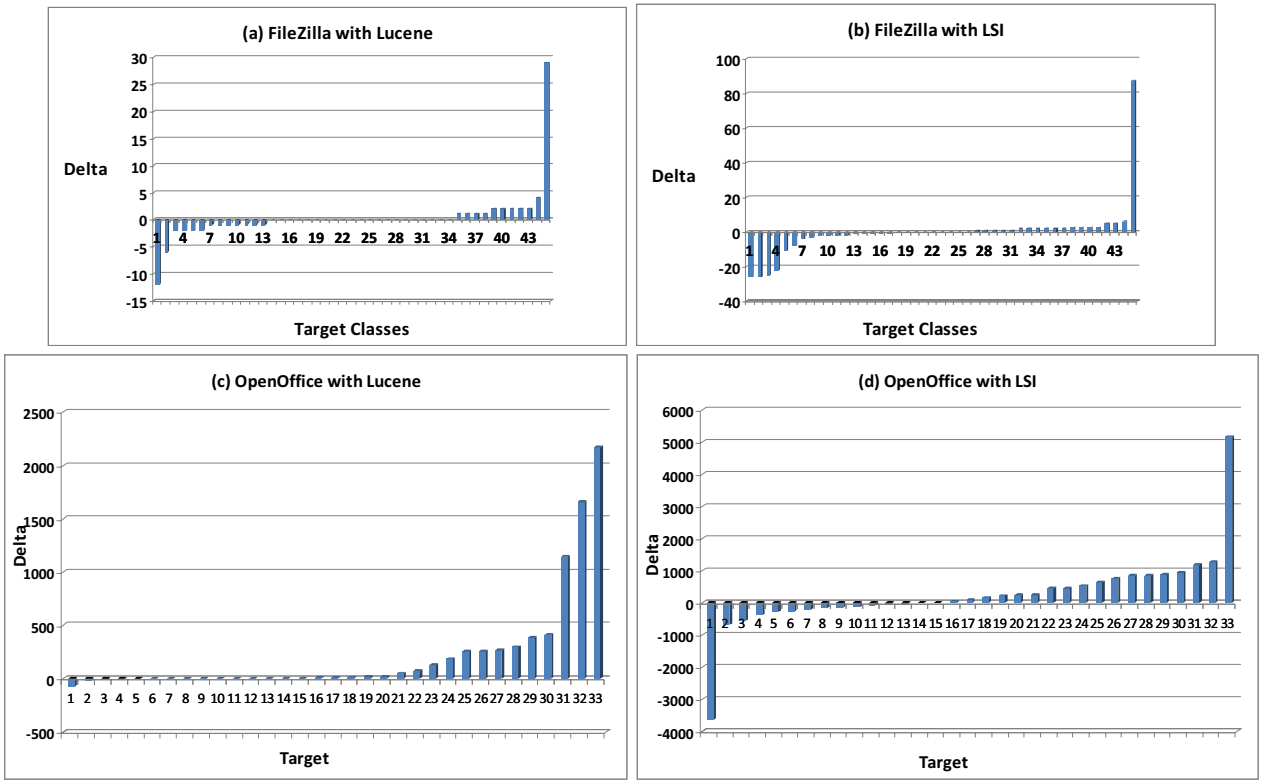
Bug: 4378 (OpenOffice)		**Bug description**	orientation of cell content gets lost if exporting as excel 97 or html. in my spreadsheet I rotated the writing in one row for 90 degrees to the left. If I export the sheet as excel 97 or html the writing is not rotated anymore.Exporting as excel 95 works fine
	Original	**Identifiers with lexicon bad smells**	bFMergeCell, bFShrinkToFit, nCIndent, nDgDiag, nGrbitDiag, nIcvDiagSer, nIReadingOrder, nTrot, ExcXf8, GetLen, GetNum
		Terms only in original corpus	xf8, excxf8, trot, ntrot, ncindent, nireadingorder, diag, ngrbitdiag, nicvdiagser, ndgdiag, bfshrinktofit, bfmergecell, num, getnum, len, getlen
	Refactored	**Refactored identifiers**	bFormatMergeCell, bFormatShrinkToFit, nCharacterIndent, nDiagonalBorderStyle, nGrbitDiagonalBorder, IndexColorValueDiagonalBorderSerial, nIndexReadingOrder, nTextRotation, ExcelFile8 GetLength, GetNumber
		Terms only in refactored corpus	**excel**, file8, excelfile8, ntextrotation, character, ncharacterindent, nindexreadingorder, diagonal, border Ngrbitdiagonalborder, serial, nindexcolorvaluediagonalborderserial, ndiagonalborderstyle, format, bformatshrinktofit, bformatmergecell, number, getnumber, length, getlength

The difference in results between the two systems can be explained by the fact that OpenOffice had a significantly worse lexicon than FileZilla, containing many more lexicon bad smells, in spite of the lower number of target classes (see Table I). In particular, OpenOffice contained many unusual abbreviations and acronyms that were expanded during refactoring, resulting in new, more expressive terms. This probably contributed in making the source code come closer to the language used in the bug descriptions and, thus, in the queries, making it easier to locate the target classes.

Another factor that could have contributed to the difference in results between the two systems is the fact that some of the types of bad smells, which were predominant in FileZilla, might have a low impact on IR-based concept location. These bad smells, i.e., *odd grammatical structure* and *inconsistent identifier use,* take into consideration the grammar and lexical form of the words found in identifiers. IR techniques, on the other hand, disregard such aspects of the identifiers, as they are purely statistical approaches. Thus, IR could be marginally impacted by the refactoring of such bad smells, which often involves changing the order of the terms in an identifier, transforming a noun in its corresponding verb, etc. While this might not have a big influence on automated tools like IR, we argue that these types of bad smells can have a significant impact on comprehension when developers are involved.

In OpenOffice, on the other hand, the most common types of bad smells were the *misspellings* and *extreme contractions.* The performance of IR can be significantly affected by these bad smells, as they can lead to the appearance of new, statistically significant terms in the corpus of classes. Thus, the good results obtained for OpenOffice after refactoring could be explained partially by the removal of these two types of bad smells.

FIGURE 1. HISTOGRAM OF DELTAS FOR FILEZILLA AND OPENOFFICE



D. *Threats to validity*

Like any case study, our study presents some threats to its validity, which we discuss in this section. First of all, generalization of the results has to be done with care. We analyzed the impact of lexicon bad smells on IR-based concept location in only two software systems, both written in C++. Having more systems, and written in other programming languages, might have led to different results.

The names proposed for the refactoring of the identifiers might have also been different if other developers would have chosen them. However, we tried to minimize this variation by following predefined rules, defined in Section II and by having two people suggest the names individually.

Last, we only identified the lexicon bad smells found in target classes and did not consider the bad smells in the rest of the source code.

V. RELATED WORK

Different approaches and tools formulated and developed to support program comprehension and maintenance tasks use identifiers as their main source of information to carry out the intended task. Some approaches [20-22] extract, split and apply natural language processing techniques on identifier names to determine the parts of speech and use them to construct graphs that support concept location tasks. Other works generate a corpus from the identifier names and apply different information retrieval techniques to support program understanding, feature and concept location tasks [14-15, 23-24, [29]]. The results of these approaches and tools depend on the quality of the identifier names. Hence, many works in the field of program comprehension and maintenance have underlined the importance of high quality, self-explanatory names, and support the consistent use of the lexicon and of naming rules in source code by identifying anomalies and suggesting improvements [2-4, 8[28]].

The quality of identifiers was studied in relationship with other code quality measures, like the number of faults in the source code, cyclomatic complexity, and readability and maintainability metrics by Butler et al. [11-12]. Boogerd and Moonen [9], on the other hand, showed that following the MISRA-C standard completely, including naming identifiers according to the prescribed conventions, would make software less reliable. Also, Arnaoudova et al. [25] showed that there is a statistically significant relationship between the entropy and context coverage of terms in identifiers, and the probability of a method being faulty.

The limitation of all these studies, however, is that they do not study the impact of bad naming on specific software comprehension tasks, and in particular on software change. The approach we presented in this paper is meant to fill in these gaps and to address the direct impact poorly named identifiers have on software change.

Empirical studies [26]-[27] have been conducted to assess the effect of code smells, such as god classes, on program understanding. Results show that the presence of more than one code smells affect program comprehensibilty

significatly [26] and that some refactorings applied to god classes allow to achieve better comprehension [27]. Both these studies focus on the effect of code smells on program understanding. In our study we instead focus on the effect of lexicon-specific smells, introduced for the first time in our previous work [1].

VI. CONCLUSIONS AND FUTURE WORK

We proposed an approach for determining the impact of lexicon bad smells in source code on software comprehension tasks performed often by developers. One example of such a task is concept location, which deals with determining the places in the source code where changes need to be implemented. We presented a case study which instantiated our approach for IR-based concept location and studied the impact of lexicon bad smells on this task in two software systems. The results indicate that lexicon bad smells can be an important factor to consider when performing IR-based concept location and that refactoring these smells can have a significant positive impact on the task. In particular, if the lexicon of the system being maintained is known to be of relatively low quality (as was the case in one of the two systems analyzed), the benefits of lexicon smell removal are expected to be quite significant.

Our future work will focus on investigating the impact of individual lexicon bad smells on concept location. At the same time, we plan to instantiate the approach introduced in this paper for other comprehension tasks and to perform empirical studies with developers, to investigate their perception of the improved code lexicon.

REFERENCES

[1] S. L. Abebe, S. Haiduc, P. Tonella, and A. Marcus, "Lexicon Bad Smells in Software," in 16th IEEE Working Conference on Reverse Engineering, 2009, pp. 95-99.

[2] F. Deissenboeck and M. Pizka, "Concise and Consistent Naming," Software Quality Journal, vol. 14, pp. 261-282, 2006.

[3] D. Lawrie, H. Feild, and D. Binkley, "An empirical study of rules for well-formed identifiers," Journal of Software Maintenance and Evolution, vol. 19, 2007.

[4] B. Caprile and P. Tonella, "Restructuring program identifier names," in International Conf. on Software Maintenance, 2000, pp. 97-107.

[5] N. Anquetil and T. Lethbridge, "Assessing the Relevance of Identifier Names in a Legacy Software System," in Annual IBM Centers for Advanced Studies Conference, 1998, pp. 213-222.

[6] T. J. Biggerstaff, B. G. Mitbander, and D. Webster, "The concept assignment problem in program understanding," in International Conference on Software Engineering, 1993, pp. 482 - 498.

[7] A. Takang, P. Grubb, and R. Macredie, "The Effects of Comments and Identifier Names on Program Comprehensibility: An Experimental Investigation," Journal of Programming Languages, vol. 4, pp. 143-167, 1996.

[8] D. Ratiu and F. Deissenboeck, "From reality to programs and (not quite) back again," in 15th IEEE International Conference on Program Comprehension, 2007, pp. 91-102.

[9] C. Boogerd and L. Moonen, "Assessing the value of coding standards: An empirical study," in International Conference on Software Maintenance, 2008, pp. 277–286.

[10] C. Boogerd and L. Moonen, "Evaluating the relation between coding standard violations and faults within and across software versions," in 6th Working Conf. on Mining Soft. Repositories, 2009, pp. 41 - 50.

[11] S. Butler, M. Wermelinger, Y. Yu, and H. Sharp, "Exploring the Influence of Identifier Names on Code Quality: an empirical study," in 14th European Conference on Software Maintenance and Reengineering, 2010, pp. 159-168.

[12] S. Butler, M. Wermelinger, Y. Yu, and H. Sharp, "Relating identifier naming flaws and code quality: An empirical study," in 16th Working Conference on Reverse Engineering, 2009, pp. 31-35.

[13] J. A. Knottnerus and F. Buntinx, The Evidence Base of Clinical Diagnosis: Theory and Methods of Diagnostic Research: John Wiley and Sons, 2008.

[14] A. Marcus, A. Sergeyev, V. Rajlich, and J. Maletic, "An Information Retrieval Approach to Concept Location in Source Code," in 11th IEEE Working Conf. on Reverse Engineering, 2004, pp. 214-223.

[15] G. Gay, S. Haiduc, A. Marcus, and T. Menzies, "On the Use of Relevance Feedback in IR-Based Concept Location," in IEEE International Conf. on Software Maintenance, 2009, pp. 351-360.

[16] D. Poshyvanyk and A. Marcus, "Combining Formal Concept Analysis with Information Retrieval for Concept Location in Source Code," in 15th IEEE International Conference on Program Comprehension, 2007, pp. 37-46.

[17] C. Jensen and W. Scacchi, "Discovering, Modeling, and Reenacting Open Source Software Development Processes," New Trends in Software Process Modeling Series in Software Engineering and Knowledge Engineering, pp. 1-20, 2006.

[18] V. Rajlich and P. Gosavi, "Incremental Change in Object-Oriented Programming," IEEE Software, vol. 21, pp. 62-69, 2004.

[19] G. Sridhara, E. Hill, L. Pollock, and K. Vijay-Shanker, "Identifying Word Relations in Software: A Comparative Study of Semantic Similarity Tools," in 16th IEEE International Conference on Program Comprehension, 2008, pp. 123 - 132.

[20] D. Shepherd, Z. P. Fry, E. Hill, L. Pollock, and K. Vijay-Shanker, " Using natural language program analysis to locate and understand action-oriented concerns," in 6th International Conference on Aspect-Oriented Software Development,, 2007, pp. 212-224.

[21] Z. P. Fry, D. Shepherd, E. Hill, L. Pollock, and K. Vijay-Shanker, "Analysing source code: looking for useful verb-direct object pairs in all the right places," IET Software, vol. 2, pp. 27-36, 2008.

[22] E. Hill, L. Pollock, and K. Vijay-Shanker, "Automatically capturing source code context of NL-queries for software maintenance and reuse," in 31st Int. Conf.on Software Engineering, 2009, pp. 232-242.

[23] S. K. Lukins, N. A. Kraft, and L. Etzkorn, "Source Code Retrieval for Bug Localization Using Latent Dirichlet Allocation," in 15th Working Conf. on Reverse Engineering, 2008, pp. 155 - 164.

[24] J. I. Maletic and A. Marcus, "Supporting Program Comprehension Using Semantic and Structural Information," in 23rd International Conference on Software Engineering, 2001, pp. 103-112.

[25] V. Arnaoudova, L. Eeshkevari, R. Oliveto, Y.-G. Gueheneuc, and G. Antoniol, "Physical and conceptual identifier dispersion: measures and relations to fault proneness," in International Conference on Software Maintenance, 2010, pp. 1 - 5.

[26] M. Abbes, F. Khomh, Y. Guéhéneuc, and G Antoniol, “An Empirical Study of the Impact of Antipatterns on Program Comprehension,” Proceedings of the 15th European Conference on Software Maintenance and Reengineering, 2011, Oldenburg, Germany.

[27] B. Du Bois, S. Demeyer, J. Verelst, T. Mens, and M. Temmerman “Does God Class Decomposition Affect Comprehensibility?,” IASTED Conf. on Software Engineering, 2006, Innsbruck, Austria.

[28] E. W. Host and B. M. Ostvold, “Debugging method names,” in Proceedings of the 23rd European Conference on ECOOP 2009 Object-Oriented Programming, ser. Genoa. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 294–317.

[29] W. Zhao, L. Zhang, Y. Liu, J. Sun, F. Yang, "SNIAFL: towards a static non-interactive approach to feature location," Software Engineering, 2004. ICSE 2004. Proceedings. 26th International Conference on , pp. 293- 303, 23-28 May 2004.