Empirical Software Engineering

Automatic Identifier Inconsistency Detection Using Code Dictionary

--Manuscript Draft--

Manuscript Number:	EMSE-D-14-00036R3
Full Title:	Automatic Identifier Inconsistency Detection Using Code Dictionary
Article Type:	Research Papers
Keywords:	inconsistent identifiers; code dictionary; source code
Corresponding Author:	Suntae Kim, Ph. D. Chonbuk National University Junju Si, KOREA, REPUBLIC OF
Corresponding Author Secondary Information:	
Corresponding Author's Institution:	Chonbuk National University
Corresponding Author's Secondary Institution:	
First Author:	Suntae Kim, Ph. D.
First Author Secondary Information:	
Order of Authors:	Suntae Kim, Ph. D.
	Dongsun Kim, Ph. D.
Order of Authors Secondary Information:	

Letter in Response to Reviewers' Comments

Prepared by
Suntae Kim‡,
Dongsun Kim†

‡Department of Software Engineering
Chonbuk National University, South Korea

†Faculty of Science, Technology and Communication,
Computer Science and Communications Research Unit,
University of Luxembourg
Luxembourg

December 29, 2014

Dear Editor and Anonymous Reviewers,

Thank you for your letter dated December 10, 2014, regarding our paper #EMSE-D-14-00036R2 entitled "**Automatic Identifier Inconsistency Detection Using Code Dictionary**". We would also like to thank the anonymous reviewers and appreciate their helpful and constructive reviews.

In the revised version of our manuscript, we have incorporated all the comments and suggestions of the three reviewers as best as we can. This letter includes both the reviewer comments and our responses to these comments. In these responses, we have explained changes made to the manuscript on the basis of the reviewers' suggestions. We have also mentioned the locations in our revised manuscript where the changes have been made to help the reviewers locate these revisions. Wherever possible, we have provided an excerpt of the revised part.

Sincerely yours,

The Authors

Editor's Comments

In preparing the new revision authors addressed most of the reviewers concerns. The writing style improved and comparison with another tool is now presented.

The paper quality was substantially improved, however, there are still a few points authors may want to consider prior to publication. In particular some style issues (reviewer 1) and some concern about the presented examples (see reviewer 3).
We understand that sometime it may be more a matter of taste, nevertheless, it is important that examples clearly motivate and properly set the stage for the paper.
⇒ We revised the style and example issues according to the reviewer comments. Please see the following responses.

Overall we suggest authors check the style (reviewer 1) and seriously consider comments of reviewer 3 on the paper presented examples, ambiguity and the need of such an approach.
⇒ For the Reviewer 1's comments, we revised all the style issues in Abstract, Section 1, 2, 3, and 4 as suggested. For the Reviewer 3's comments, we tried to convince why we used the examples and how our evaluation process confirmed that the examples are valid. Note that 16 human subjects reviewed the detection results and these are some of examples shown in the paper.

Reviewer #1's Comments:

The authors substantially improved the manuscript: they added a comparison with existing work, fixed a few errors, and rephrased most of the unclear sentences.

⇒ We appreciate the constructive suggestions and comments.

I would suggest a few minor improvements before publication:

- incFile(): Based on the Javadoc of the method ("Method implements the logic for filtering file name inclusion") this must be expanded to 'include' rather than 'increase' as it is currently in Table 2.

⇒ This example in Table 2 is removed from the row ("inc ⇒ increase") as the reviewer pointed out.

- "inconsistent identifiers frequently and inevitably occurs"
- "this issue can be arise"
- "We then conducted a user study to check the validity of these results. This study asked 16 developers to evaluate the precision and recall of detection results." -> "To evaluate the precision and recall of detection results we conducted a user study involving 16 developers."
- "A naming convention defines a specific way in which names identifiers in programs." -> to name?

⇒ All these typos and grammar errors are fixed accordingly.

Reviewer #2's Comments:

The authors have properly addressed my remaining comments and suggestions. In particular, they have performed a direct comparison with an existing tool, LBSDetector. So, I think the paper can now be accepted for publication.

⇒ We appreciate the suggestions and comments to improve this paper.

Reviewer #3's Comments:

The authors propose an automated aproach to detecting identifier inconsistencies in source code. They automatically build a code dictionary from the existing API documents of popular Java projects by using a Natural Language Processing (NLP) parser. The approach then takes a target program and detects inconsistent identifiers of the program by leveraging the code dictionary. They provide CodeAmigo, a GUI-based tool support for their approach. They evaluated their approach on seven Java based open-/proprietary- source projects. They attained precision and recall > 80%.

Overall the authors propose a solution to an very interesting and important problem. However, I am not convinced entirely of the approach and the novelty seems rather weak. In many cases, the examples given throughout the paper unfortunately serve to illustrate the weakness of this submission.
⇒ We stated our sincere responses for each comment. These responses may convince the reviewer.

For example, consider the 2nd paragraph on page 23 in Section 1. The authors say that a developer would be confused between "makeObject" and "createObject". Who would be confused by this? It is just a normal synonym usage that occurs in every day life as in "remove item"/"delete item". I honestly don't think anybody would be confused by such synonym usages. Also consider "create jobs"/"make jobs", the meaning is very clear although the latter is probably an awkward phrase formation, possibly grammatically incorrect even. However, most developers who read such code and most people who read such English text, would "get" the idea.
⇒ Our focus is clearly different from what the reviewer assumed. First, if the two identifiers, "makeObject" and "createObject", are used in different programs, they have no problem with understanding or using the programs for developers. However, they would easily lead to a confusion if they are used in a single program; what if those identifiers are used in a single method, class, or package? This is also confirmed by the interview results described in Section 4.3.4 and several real examples throughout the paper. It is obvious that developers are trying to avoid the use of inconsistent identifiers. Although a program writer might define her/his own concept for each synonym (e.g., "make*" and "create*"), several program readers may misunderstand the concepts. Thus, this paper focuses on detecting inconsistent identifiers containing synonyms and grammar errors throughout a project.

I find it extremely hard to believe that some developer would get confused between "status" and "states" (same pararaph as above). You really need to back such claims with a survey and so on.
⇒ As stated above, if these identifiers are used in a single method, it can make a confusion. In particular, new developers may need more time to understand the difference of two words and it may decline productivity. For this case, we already added real examples in Footnote 2 and 3.
[Footnote 2] http://www.dlib.vt.edu/projects/MarianJava/edu/vt/marian/server/status.java
[Footnote 3] https://github.com/tangmatt/word-scramble/blob/master/system/Status.java

Consider the semantic consistency, "fetch" and "get" on page 44. Why is it a semantic inconsistency? There is a subtle difference in meaning here. "Get" means to simply return the value of a field as in the following example:

```
public int getLength() {return length;}.
```

Typically "get" methods will be a one line method as shown in my example.

However, "fetch" may be a more involved operation as in "fetching" records from a database. This will not be a one line method but many lines of code will be involved such as establishing a connection to the database, issuing an SQL query, reading results and so on.

"Selector" and "Chooser" may be semantically inconsistent, but is it a problem?

⇒ These are the same issues with the abovementioned problems. Although some developers are using those words ("get" and "fetch") in a certain way, there is no clear consensus that "get" should be used in this way and "fetch" in that way. Even if the author of the program defined his/her own way to use words in identifiers, it is still vulnerable for (new) program readers to be confused by the two similar synonyms.

The "word-pos" inconsistency is even more baffling - yes, DrawApplet (page 44) is probably mis-named, it would have been better named as "AppletDrawer" but is it a problem for a human? It may pose a problem for tools such as code summarizers but for humans, I am not sure if it is even a problem.

⇒ The word-POS inconsistency is defined for detecting the violation of Java naming convention. As described in Section 4.3.4, developers often point out that the naming convention violation may impose a maintenance burden if several developers as a team work together on writing the program. Suppose that each developer uses words to name identifiers in different ways. Then, these developers may not clearly understand the identifiers created by other developers. Consequently, this can be a disaster for a newly coming developer. In addition, to follow the convention, "DrawApplet" should be revised as "DrawerApplet".

The "readerIndex" on page 43 is another example of a "poor example". The code that uses this identifier would probably look like this (at the call site):

```
int ri=readerIndex(...);
```

The code context makes it abundantly clear that this method is a "getter". Yes, it would have been better to name this method as "get"ReaderIndex, but "get" has been omitted for brevity. So, this may be an inconsistency, but not a problem.

See the documentation for "readerIndex" at https://lucene.apache.org/core/4_3_0/core/org/apache/lucene/index/BaseCompositeReader.html#readerIndex%28int%29. The documentation makes it very clear that the method is a "getter"

---"Helper method for subclasses to get the corresponding reader for a doc ID".

⇒ Basically, this is detected since it is a naming convention violation. It is obvious that readers can get the idea of this method if she/he reads the API document. However, it would be better if it is intuitively understandable. In addition, it is easy to find out the meaning of the identifier of a method if the method is already used in a program since it can provide additional context information. On the other hand, this is not applicable when newly using the method. Thus, it is important to avoid the use of inconsistent identifiers as well as naming convention violation.

"startsWith" is shown to be semantically inconsistent with "start". Why? "Start" is typically a method that "starts" something, for ex: "start a thread". "startsWith" is a boolean checker method as in does this paragraph start with the letter "S"?. So I would say that both of them are not related at all, but orthogonal in meaning. It is incorrect to say "startsWith" is semantically inconsistent with "start".

⇒ First, this is “syntactical inconsistency” rather than “semantic inconsistency”.

- startsWith() (method, JMeter) - ‘starts’ is syntactically similar to start(starts is used 1 times; start is used 45 times)”.

Thus, our approach does not consider the meaning of both identifiers (or words) here. Rather, the approach detected those identifiers since they can be syntactically too close (i.e., too similar spelling). This can be an issue when refactoring the source code or commencing “find and replace”.

Thus, to conclude my analysis of the examples shown (in introduction and section 4), the authors need to understand that an "inconsistency" per se may not be a problem. Further some of the inconsistencies shown are NOT inconsistencies at all!

⇒ All examples shown in this paper are reviewed and confirmed by real developers throughout our evaluation process as described in Section 4. These developers stated that the identifiers detected by our approach were inconsistent and may impose additional maintenance burdens later. This implies that the examples can show the motivation of this paper and the effectiveness of our approach.

In Section 4, Evaluation, I am not convinced of the explanations given behind choosing the 7 open source projects. Also why not run the experiments on a larger set?

⇒ Note that the manual evaluation of inconsistent identifiers costs a lot. Our approach detected 3,826 inconsistent identifiers out of 55,526 identifiers in the seven projects. Even reviewing this number of identifiers may impose too much burden to each human subject. Evaluating more than that can make fatigue problems and inaccurate results. This is why we selected one subject (JUnit) for computing the recall value in Section 4.3.2.

In evaluation on page 45, a lengthy explanation is given, but it does not answer the question, "how often did the evaluators actually access the source code in the evaluation?". The authors in their response to the original reviews state that they did not collect this statistic. They really should.

⇒ Basically, our approach and its evaluation were not designed to compute the number of accesses to the source code. Although counting the number would be helpful to understand the developers' behavior, this is not our focus in this study. We could figure out that the human subjects frequently accessed the source code throughout the interview described in Section 4.3.4. In addition, counting the accesses may need to build another plug-in to capture the developers' code browsing behaviors. This could impose another cost in the evaluation process.

The 2nd reason on page 46 for not using AUC but sticking with Precision-Recall is really confusing. I understand it as saying that using AUC etc required too much work and hence was not done. If my understanding is incorrect, please forgive me, but if you are really saying that it involves a lot more work, then it should be done and is not a valid reason to not use AUC.

⇒ Computing ROC and its AUC may lead to a fatigue problem. Suppose that each developer should manually evaluate and review more than 50,000 identifiers to figure out whether each identifier is inconsistently used or not; this makes the human subjects quickly tired and may affect the correctness of inconsistency evaluation. Since ROC and AUC require recall values, this process is necessary. However, it takes too much cost and time to let the developers focus on reviewing 50K identifiers. This is also described in Section 4.3.1 as follows:

> In order to evaluate the validity of our approach, we applied a traditional precision and recall measure [31] instead of measures such as the area-under-ROC curve [40] with the following reasons. First, it is almost impossible to manually find true-negative identifiers for the 7 projects' source code. Second, we have defined the thresholds in the preliminary study, and changing the thresholds means that all manual evaluation processes should be conducted from the start to obtain a new confusion matrix. Third, we considered that the precision and recall measure is sufficient to explain the efficiency of our approach, since true-negative detections are not considered for the precision measure.

On page 54 in the section "Threats to validity", the authors acknowledge "Contextual information can alleviate inconsistency issues." They claim that "Evaluating the impact of the contextual information with inconsistency detection remains as a future task"

In my opinion, contextual information plays a very significant role and can help resolve many inconsistency issues. Thus, the impact of contextual information MUST be evaluated in the same paper that proposes an approach to detecting inconsistent identifiers. It could so happen that a future study clearly confirms that contextual source code information can help resolve any inconsistencies and thus obviate the need for the approaches suggested in this paper! Thus, it should NOT be future work but current work which is a part of this submission.

⇒ In fact, our approach implicitly uses another context information in a different way: domain words and idioms collected from several popular Java projects. The Reviewer #2's comment in the previous review results is about the context information for method identifiers such as parameter names. As stated in Section 4.4, we consider this issue another independent topic and are preparing another study for this topic.

NOTE: The page numbers that I use refer to the "66 page" pdf that I received. Thus, for example, pages 43, 44 and 45 that I use refer to pages 22,23 and 24 of the actual manuscript.

Noname manuscript No.
(will be inserted by the editor)

Automatic Identifier Inconsistency Detection Using Code Dictionary

Suntae Kim · **Dongsun Kim***

Received: date / Accepted: date

Abstract Inconsistent identifiers make it difficult for developers to understand source code. In particular, large software systems written by several developers can be vulnerable to identifier inconsistency. Unfortunately, it is not easy to detect inconsistent identifiers that are already used in source code. Although several techniques have been proposed to address this issue, many of these techniques can result in false alarms since such techniques do not accept domain words and idiom identifiers that are widely used in programming practice. This paper proposes an approach to detecting inconsistent identifiers based on a custom code dictionary. It first automatically builds a Code Dictionary from the existing API documents of popular Java projects by using an Natural Language Processing (NLP) parser. This dictionary records domain words with dominant part-of-speech (POS) and idiom identifiers. This set of domain words and idioms can improve the accuracy when detecting inconsistencies by reducing false alarms. The approach then takes a target program and detects inconsistent identifiers of the program by leveraging the Code Dictionary. We provide *CodeAmigo*, a GUI-based tool support for our approach. We evaluated our approach on seven Java based open-/proprietary-source projects. The results of the evaluations show that the approach can detect inconsis-

S. Kim
Dept. of Software Engineering, Chonbuk National University, 567 Baekje-daero, Deokjin-gu, Jeonju-si, Jeollabuk-do 561-756 Republic of Korea
Tel.: +82-63-270-4788
Fax: +82-63-270-4767
E-mail: stkim@jbnu.ac.kr

D. Kim(*Corresponding Author*)
Faculty of Science, Technology and Communication, Computer Science and Communications Research Unit, Interdisciplinary Centre for Security, Reliability and Trust, University of Luxembourg, 4 rue Alphonse Weicker, L-2721, Luxembourg
Tel.: +352 46 66 44 5573
Fax: +352 46 66 44 5356
E-mail: dongsun.kim@uni.lu

tent identifiers with 85.4% precision and 83.59% recall values. In addition, we conducted an interview with developers who used our approach, and the interview confirmed that inconsistent identifiers frequently and inevitably occur in most software projects. The interviewees then stated that our approach can help to better detect inconsistent identifiers that would have been missed through manual detection.

Keywords inconsistent identifiers · code dictionary · source code

1 Introduction

The use of consistent identifiers (such as class/variable/method names) is one of the key aspects for understanding source code. Developers heavily rely on identifiers, which cover about 70% of the source code elements, to understand programs [1]. However, inconsistent identifiers can impede program comprehension [2] and can thus hinder software maintenance, which is a very expensive activity in software development. Inconsistent identifiers can also negatively impact automated tools that were designed to help program understanding and maintenance such as automated code summarization techniques, feature location techniques, etc.

For example, suppose that developers have made the two methods `makeObject()` and `createObject()`[1]. Readers of the identifiers might be confused due to similarity between the meaning of the words *make* and *create* regardless of the developers' intentions. In addition, this issue can arise in the use of terms `states` and `status` which have similar letter sequences but have different meanings[2,3]. These inconsistencies may cause potential maintenance problems or defects in a program.

In the software industry, a large number of practitioners strongly emphasize the need for identifier consistency. For example, Martin [3] proposed for programmers to use a single word per concept and to have consistency throughout the entire project. He also stated that "a consistent lexicon is a great boon to the programmers who must use your code" [3]. He pointed out code readers often rely on names in the source code to understand programs. Unfortunately, most programs suffer from inconsistent identifiers since contemporary software projects are mostly developed by a large group of developers. In addition, a long revision history can lead to inconsistency as well [4]. As new developers take over legacy programs, they often make use of identifiers that are inconsistent with those in the preexisting code.

Several techniques have been developed to detect inconsistent identifiers. One stream of research has presented techniques that can map identifiers into their intended concepts through manual or automatic mechanisms to find inconsistent identifiers [1,2,5,6]. However, these mechanisms focus only on semantic inconsistency without considering diverse POS usages of each word. Another stream has established a set of vocabulary or ontology derived from source code. While these techniques proactively help a developer to write a code with

[1] http://goo.gl/p6Gzmd and http://goo.gl/7cCV8n
[2] http://www.dlib.vt.edu/projects/MarianJava/edu/vt/marian/server/status.java
[3] https://github.com/tangmatt/word-scramble/blob/master/system/Status.java

consistent identifiers [7–12], they do not detect inconsistent use of identifiers within the actual source code of a single project.

This paper presents an approach that detects inconsistent identifiers based on a custom Code Dictionary that has been enhanced through our previous work [13]. This approach discovers inconsistent code names using natural language processing. In order to improve the precision of the detection, we first build a Code Dictionary by scanning the application programming interface (API) documents of 14 popular Java projects. This dictionary defines the domain words according to their POS and idiom words in order to understand how words are used in programming. Such techniques help to avoid false alarms. Based on the Code Dictionary, our approach then takes the source code of a program as an input, scans all the identifiers, and discover inconsistent identifiers. In this paper, three types of inconsistencies are detected: semantic, syntactic, and POS inconsistencies. These respectively represent 1) two different words that indicate the same concept, 2) two words that have similar letter sequences, and 3) a word used as an inconsistent POS. This paper also introduces the three types of inconsistencies by revising concepts and definitions presented by existing work.

To evaluate our approach, we first carried out a preliminary sensitivity analysis of the thresholds in order to detect more inconsistent identifiers. Then, we applied it to seven open/proprietary source projects to detect inconsistent identifiers. The approach detected 3,826 inconsistencies[4] from 55,526 identifiers collected from the projects. To evaluate the precision and recall of detection results, we conducted a user study involving 16 developers. The result of our study shows that our approach detected inconsistent identifiers with a precision of 85.4% and a recall of 83.59%. Also, we carried out a semi-structured interview with developers in order to investigate the usefulness of our approach. They stated that there are many inconsistent identifiers and that these make programs difficult to understand. However, inconsistency is also difficult to detect manually. Therefore, our approach might be useful for identifying inconsistent identifiers to make a program easier to understand.

Our contributions can be summarized as follows:

1. **Inconsistency detection based on a Code Dictionary:** We present a novel approach for automatic inconsistent identifier detection that leverages the Code Dictionary which defines the domain words and idioms in order to reduce false alarms.
2. **Empirical evaluation:** We present the results of an empirical evaluation by applying our approach to seven open/proprietary source projects.

All materials used in this paper and for the results of the detailed experiment are publicly available at our project website[5].

The remainder of the paper is organized as follows: Section 2 presents the background on Java naming convention and on identifier inconsistency. Section 3 introduces our approach for detection of inconsistent identifiers in source code. Section 4 presents the preliminary study and the three-step evaluation of the proposed approach. After discussing a set of related work in Section 5, we conclude this paper and also discuss future work in Section 6.

[4] Note that an identifier can include multiple inconsistencies. The total number of unique identifiers containing at least one inconsistency is 1,952.

[5] https://sites.google.com/site/detectinginconsistency/

2 Background

This section presents Java naming conventions that provide guidance on how to name identifiers, and then formulates several types of inconsistencies frequently discovered in source code.

2.1 Java Naming Convention

A naming convention defines a specific way to determine identifiers in programs. These conventions are useful when several developers work collaboratively to write a program while attempting to maintain consistency in the identifiers since this can make the source code easier to understand. The Java naming convention published by Sun Microsystems (acquired by Oracle) [14] introduces common naming guidelines of Java identifiers. According to these guidelines, all identifiers should be descriptive and should observe the following grammatical rules, depending on the types:

- **Classes and Interfaces** should be nouns (a noun phrase) starting with a capital letter.
- **Methods** should be verbs (a verb phrase) and should start with a lowercase.
- **Attributes** and **Parameters** should be a noun phrase with a lowercase first letter.
- **Constants** should be a noun phrase with all letters as upper cases separated by underscores.

In addition, composite identifiers should be written in a camel case (mixed with upper and lower cases). For example, a class identifier *WhitespaceTokenizer* is a noun phrase of two words: *Whitespace* and *Tokenizer*. The method identifier *getElementForView()* can be split into *get*, *Element*, *For* and *View*, which compose a verb phrase with a prepositional phrase.

Note that programmers in particular tend to extensively rely on naming conventions for identifiers in order to write more readable source code in Java programs [15]. This implies that the use of consistent words in program identifiers is important for software maintenance. Suppose that a new developer comes in to collaborate on a software project. To understand how it works, she/he needs to read a part of the source code. Reading the method and the attribute names usually helps to gain an understanding. On the other hand, what if different words are used in parameter names that indicate the same concept? What if a single word is used for many different concepts? This makes the program difficult to read.

2.2 Three Types of Inconsistent Identifiers

This section formulates three inconsistency types: semantic, syntactic, and POS, based on the concepts presented by previous work. First, the semantic inconsistency indicates the use of diverse synonyms in multiple identifiers. For example, for the class names, `LdapServer` and `LdapService` recoded in Issue [DIRSERVER-1140][6], `Server` and `Service` are different words, but they imply a similar meaning regardless of the programmer's intention. The issue submitter (and patch creator as well) stated that this inconsistency was propagated to another plug-in (`ApacheDS` plug-ins for `Studio`) via a resource file (`server.xml`) and in the documentation as well. Even if a writer distinguishes the two words based on their definition consistently throughout the project [16], program readers may not precisely catch the slight difference between the words and this can eventually result in a misunderstanding. Similar issues are observed in a wide range of programs, such as in a pair of `Real` and `Scala` described in Issue [Math-707][7]. In addition, this inconsistency is common in many software projects in which many developers are involved [17].

The semantic inconsistency includes the concept *synonyms* as defined in [1,2]. Additionally, it sets constraints for the two words which should be of the same POS. This constraint originates from the definition of a *synonym*, which is that, "synonyms belong to the same part of speech." [8]. In addition, dictionaries such as Oxford [9], Collins Cobuild [10], Dictionary.com [11] and WordNet [18] classify synonyms in terms of the POS. When searching for synonyms, adding the POS constraints in the definition contributes to an effective reduction in the search space from all possible POSes to one specific POS. In WordNet, for example, the word *use* in WordNet has 17 synonyms as a noun, and 8 synonyms as a verb. The POS constraints reduce the search space from 25 to 8 if the POS is recognized as a verb.

The following definitions formulate the semantic inconsistency:

Definition 1 A word set W is defined as a collection of any finite sequence of the English alphabet.

Definition 2 C is a set of concepts.

Definition 3 An identifier set ID is defined as a collection of any finite token sequence of $w \in W$ and literals (digits $DIGIT$ and special characters SC). For example, $id \in ID$ can be $(t_1 t_2 t_3 \ldots t_N)$ where $t_i \in T = W \cup DIGIT \cup SC$. Its index function is defined as $f_i : ID \times \mathbb{N} \rightarrow T$.

Definition 4 A tagging function f_t is defined as $f_t : ID \times \mathbb{N} \rightarrow POS$ where POS is a set of {noun, adjective, verb, adverb, preposition, conjunction, non-word terminal}. For example, $f_t(\text{"setToken"}, 1) = f_t(t_1 = \text{"set"}) = verb$.

Definition 5 A concept map[12] D is defined by a map $D : W \times POS \rightarrow 2^C$.

[6] `Apache Directory Project`: https://issues.apache.org/jira/browse/DIRSERVER-1140
[7] `Apache Commons Math`: https://issues.apache.org/jira/browse/MATH-707
[8] Synonyms Definition:http://en.wikipedia.org/wiki/Synonym
[9] Oxford Dictionary, http://www.oxforddictionaries.com/
[10] Collins Cobuild Dictionary:http://www.collinsdictionary.com/dictionary/english
[11] Dictionary.com:http://dictionary.reference.com/
[12] To define this map, any English dictionary can be used. In this paper, we used WordNet [18] as described in Section 3.2.2.

Definition 6 (**Semantic Inconsistency**) Two identifiers, id_1 and id_2, have *semantic inconsistency* if $\exists w_1 = f_i(id_1, i)$, $w_2 = f_i(id_2, j)$, and $D(w_1, tag) = D(w_2, tag)$ where $tag \in POS$ and $w_1 \neq w_2$.

Second, syntactic inconsistency occurs when multiple identifiers use words with a similar letter sequence. The identifier pairs `getUserStates()` and `getUserStatus()`, `ApplicationConfiguration` and `ApplicationConfigurator`[13], `memcache` and `memcached`[14], are examples of this inconsistency. Code readers might be confused by a pair of words that seem identical due to having 1) similar length, 2) small edit distance, and 3) long sequences. Haber & Schinder [19] and Monk & Hulme [20] performed research that concludes that readers are more susceptible to confusion if words have the similar shapes, including word-length, especially for long words. Writing source code is similar to general writing, especially for writing words. This is also applicable to source code [3]. Syntactic inconsistency is caused by typos and inconsistent use of singular/plural forms in naming methods. In addition, syntactically inconsistent identifiers are commonly generated by unfaithful naming of variables (e.g., `arg1`, `arg2`, `param1` and `param2`).

Syntactic inconsistency may result in maintenance problems, where code readers can misunderstand, particularly when syntactically inconsistent words are discovered in the dictionary (e.g., `states` and `status`) because their meanings are clearly different. In addition, when automatically renaming words using "find and replace", the mechanism would not work properly, since spell checkers may not work when these identifiers could have a valid in spelling. Code generation is another example, e.g., Issue [Bug 108384][15] of Eclipse describes how similar names used in a program can lead to duplicate identifiers when generating another program based on the source code.

We defined syntactic inconsistency as follows:

Definition 7 (**Syntactic Inconsistency**) Two identifiers, id_1 and id_2, are *syntactically inconsistent* if $\exists w_1 = f_i(id_1, i)$, $w_2 = f_i(i_2, j)$, and $C(w_1, w_2) = \frac{max\{L(w_1), L(w_2)\}}{(|L(w_1) - L(w_2)| + 1) \cdot DIST(w_1, w_2)} > K$. K is *closeness threshold* and *DIST* computes an edit distance between two words and is defined as $DIST : W \times W \rightarrow \mathbb{Z}^+$. L counts the number of letters in a word and is defined as $L : W \rightarrow \mathbb{N}$. $C(w_1, w_2)$ is not defined if $DIST(w_1, w_2) = 0$.

The above definition implies that human developers can become more confused if two long words have a small edit distance and similar word length [19, 20]. Note that this definition does not include the exception where noun words have the same root after stemming, for example, `accent` and `accents`.

Third, POS inconsistency implies that identifiers use homonyms or violate naming conventions. There are two sub-types of POS inconsistencies: 1) Word-POS inconsistency and 2) Phrase-POS inconsistency. Word-POS inconsistency happens when the same word is used for different POSes in multiple identifiers. For example, the word *short* in the method identifier

[13] https://bugs.eclipse.org/bugs/show_bug.cgi?id=369942
[14] https://github.com/Chassis/memcache/issues/2
[15] https://bugs.eclipse.org/bugs/show_bug.cgi?id=108384

getShortType() and in the class identifier *ShortName* is respectively used as a noun denoting a short data type and as an adjective.

Caprile and Tonella [21] observed that developers tend to consistently use a single POS of a word throughout a project even when the word can have diverse POSes. For example, the word *free* can be used as a verb, adjective, and adverb in natural language while only the use as a verb of the word was observed in a specific software project. One of the real cases for this inconsistency is shown in an issue report[16] that indicates that the word *return* in a variable name `returnString` can be confusing since `return` is often used in a method identifier as a verb. Consequently, the corresponding patch[17] changes `returnString` to `resultString`.

Definition 8 (**Word-POS Inconsistency**) Two identifiers, id_1 and id_2, are *Word-POS inconsistent* if $\exists w = f_i(id_1, i) = f_i(id_2, j)$ and $f_t(id_1, i) \neq f_t(id_2, j)$.

Phrase-POS inconsistency occurs when identifiers violate the grammatical rules of a Java naming convention. For example, when *Aborted* and *Restrict* are used as class identifiers, these are inconsistent with the naming convention since they are an adjective and a verb, respectively. Similarly, when *directory()* and *newParallel()* are used as method identifiers, they violate Phrase-POS consistency since they should be verb phrases.

This convention is shown in the discussion of another issue report[18], where developers indicate they prefer to conform to POS conventions. For example, between `getFirst()` and `first()`, more developers in the discussion chose the former since the verb prefix can clarify the meaning of the method.

Definition 9 (**Identifier Type & Phrase-POS Rules**) $\forall i \in ID$, type function $f_{type} : ID \rightarrow TYPE$ defines i's identifier type where $TYPE = \{class, method, attribute\}$. $R_{POS} : TYPE \rightarrow POS$ defines Phrase-POS rules and $f_{POS} : ID \rightarrow POS$ determines the Phrase-POS of an identifier.

Definition 10 (**Phrase-POS Inconsistency**) An identifier $i \in ID$ is Phrase-POS inconsistent if $f_{POS}(i) \neq R_{POS}(f_{type}(i))$.

2.3 Challenges

To detect the aforementioned inconsistencies, the following issues should be addressed.

2.3.1 POS Usage

Since the inconsistency detection that is described in Section 2.2 depends on the POS usage, the method becomes inaccurate if we cannot extract the POS information from the identifiers. Most contemporary NLP parsers [22] [23] can identify the POS usage of the words

[16] https://github.com/scrom/Experiments/issues/32
[17] https://github.com/scrom/Experiments/commit/04dfbf7818626f9818379eb20e4c87e755407687
[18] https://github.com/morrisonlevi/Ardent/issues/17

used in a sentence. However, the method can become confused since several words in the source code are used in different POS when compared to POS usage in natural languages. Thus, identifying the POS usage of words used in programs is necessary for inconsistency detection.

2.3.2 Domain Words

In natural languages, some words can be used as different POSes, but computer programs tend to use a word as a single POS [21]. For example, the word 'file' is frequently used as a noun or as a verb in natural languages. However, it is mostly used as a noun denoting a notion for data storage in the computer domain. Similarly, words such as 'default', 'value', and 'input' are generally used as a noun, even they are often used as several different POSes in natural languages. If we can figure out the dominant POS of each word in advance, then the detection of inconsistent identifiers can improve.

2.3.3 Idiom Identifiers

Some inconsistent identifiers can be accepted as an exception even if they violate the grammatical rules of the naming conventions. For example, the method identifiers *size()*, *length()* and *intVal()* used in the Java Development Kit (JDK) or in popular projects do not observe the grammatical rules but their use is widely accepted in Java programs.

In addition, several words are commonly abbreviated in the computer science domain. For example, 'spec', 'alloc', 'doc' are used instead of 'specification', 'allocation' and 'document'. However, NLP parsers cannot recognize whether they are abbreviated words or not. This decreases the accuracy of the parsed results.

We define the idiom identifiers for each type of inconsistency. For semantic inconsistency, although word w_1 has a conflict with word w_2 (i.e., $D(w_1,tag) = D(w_2,tag)$ for an arbitrary tag), w_1 is not detected as an inconsistency if w_1 is defined as an idiom. Similarly, words with syntactic inconsistency can be also accepted. With respect to a POS inconsistency, we can skip the consistency check if an identifier is included in the idiom set.

2.3.4 Tool Support

It might be difficult for developers to navigate and examine a list of inconsistent identifiers without GUI-support if there is a large number of inconsistencies. A tool support can alleviate this burden, and this tool should be integrated into existing development environment so that developers can easily find and correct inconsistent identifiers.

3 Approach

This section presents our approach for detecting identifier inconsistency. Figure 1 shows the overview of our approach. This approach has two phases: 1) building a code dictionary, and

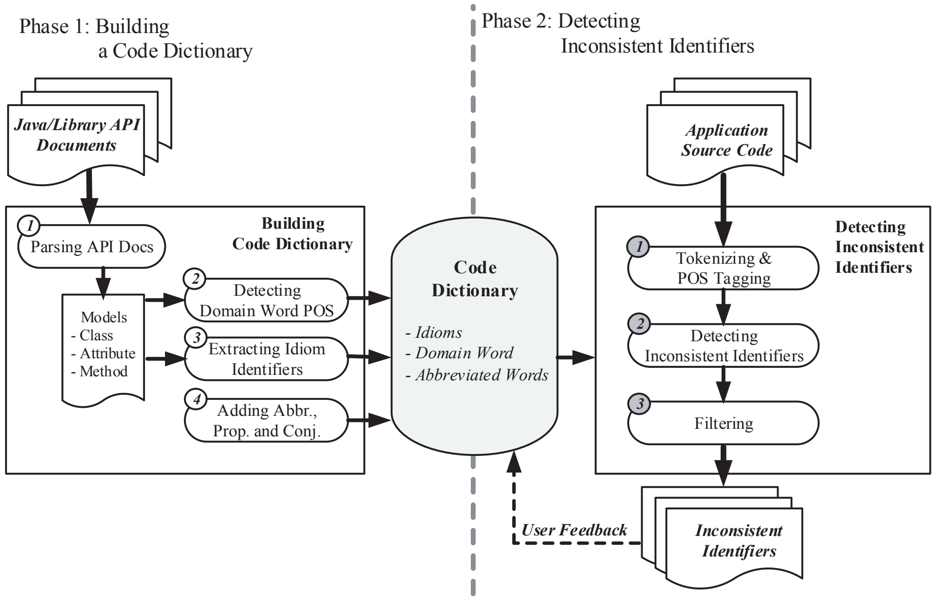


Fig. 1: Overview of our approach to inconsistent identifier detection.

2) detecting inconsistent identifiers. In the first phase, this approach analyzes the API documents that are trusted by the users and collects the words that are necessary to build a Code Dictionary. This dictionary extracts the domain words with dominant POSes, idioms, and abbreviated words from the trusted API documents. The second phase scans a program and finds the inconsistent identifiers. The Code Dictionary built in the first phase can reduce the frequency of false alarms by filtering out domain words and idioms from the set of detected identifiers. The remainder of this section explains the details of our approach.

3.1 Phase 1: Building a Code Dictionary

Our approach first creates a Code Dictionary since common English dictionaries may not sufficiently deal with identifiers in programs. For example, there exist many idioms and domain-specific words in the source code of programs, such as *file* and *rollback*. Ordinary NLP parsers often result in confusion during POS tagging. The Code Dictionary is basically a customized dictionary that maps a word to its POS using exceptional rules. The dictionary is used to filter out wrong parsing results from the NLP parser in order to increase the precision of detections. In this phase, this approach first parses the API documents of programs trusted by a user and collects code identifiers from the documents. Then, it discovers the POSes of the words used in the collected identifiers. Based on the POS discovery results, the approach collects the idioms, domain, and abbreviated words to build a Code Dictionary. Note that this dictionary is built once and can be reused several times.

To build a Code Dictionary, the user can designate his/her own trusted documents. Basically, this approach collects class, method, and attribute identifiers from 14 API documents, as shown in Table 1. The user can add other API documents or remove existing documents in order to customize the Code Dictionary.

3.1.1 Parsing API Documents and Recognizing POSes

This approach leverages an NLP parser[19] to parse an identifier as a sentence. The result of the parsing is used to collect the POS information of each word in the identifier. The POS information is important since it helps in the detection of domain words, as described in Section 2.3.2.

Our approach first collects the code identifiers in the API documents. The identifiers are tokenized according to the rule of the camel case, also using an underscore as a separator. For the method identifiers, a period is inserted at the end of the last term to make a complete sentence since the method identifiers constitute a verb phrase, which is a complete sentence without a subject. This often helps the NLP parsers to recognize the POSes of words with greater precision. For example, the identifier of the *getWordSet()* method is converted into the phrase, 'get Word Set.'. Then, the NLP parser analyzes POSes of each word and phrases within the sentence, resulting in '[(VP (VB get) (NP(NNP Word)(NNP Set)(..)))]', where VP, VB, NP and NN denote a verb phrase, verb, noun phrase and noun respectively. These tags are used according to the Penn Treebank Project [28].

3.1.2 Identifying Domain Words

The process used to identify domain words and their major POSes consists of two filters: a *W-Occurrence Filter* and a *POS Ratio Filter*. In the first filter, words occurring less than $T(WO)$ in the trusted API documents are filtered out from all the candidate words, where $T(WO)$ is the threshold of the occurrence of a word. This is because domain words are used more frequently than other non-domain words in code identifiers. The second step determines the major POS of each domain word. If any POS out of all POSes available for a word accounts for more than $T(PR)$, the POS is regarded to be the dominant POS, as described in Section 2.3.2, where $T(PR)$ indicates the threshold ratio used to decide the dominant POS of a word. If a word passes these two filters, the word is then collected as a domain word in the Code Dictionary. We have defined the two thresholds, $T(WO)$ and $T(PR)$, by conducting a preliminary study before the full evaluation (see Section 4).

[19] Although there are some of the researches on POS-tagging of source code elements [9, 24, 25], they are not publicly available or also used natural language parser such as Minipar [26], Stanford Log-linear Part-Of-Speech Tagger [27]. In this paper, we have adopted Stanford Parser [22] because it is highly accurate for parsing natural language sentences and broadly used for NLP. In addition, it is publicly available, well-documented and stable.

Table 1: Java/Library API documents for building a code dictionary.

Library	Version	Description
Java Development Kit	1.7.0	Java standard development kit
Apache Ant	1.8.3	Java library and command-line tool for building java project
Apache POI	3.9	APIs for manipulating various file formats based upon Microsoft
Apache Commons-Lang	3.3.1	Collections of extra methods for standard Java libraries
Apache Commons-Logging	1.1.1	Bridge library between different logging implementations
Apache Commons-Collections	4.4	Enhanced library of Java collections framework
Apache JMeter	2.9	Java application for load test and measure performance
Java Mail	1.4.5	Framework to build mail and messaging applications
JDOM	2.0.5	Library for accessing, manipulating, and outputting XML data
JUnit	4.10	Framework to write repeatable unit tests
Apache Log4J	1.2.17	Logging framework library for Java
QuaQua	1.2.17	User Interface library for Java Swing applications
StanfordParser	3.2.0	Library that works out the grammatical structure of sentence
Joda-time	2.2	Java date and time API

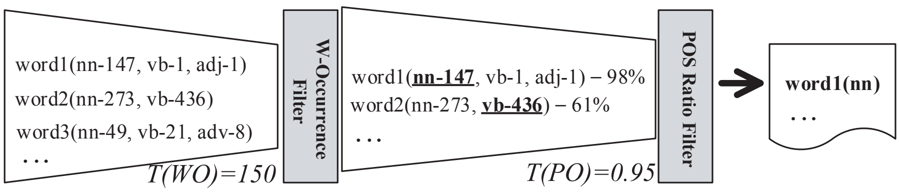


Fig. 2: Domain word identification. *nn*, *vb*, *adj*, and *adv* denote a noun, verb, adjective and adverb, respectively.

Figure 2 shows an example of how domain words are collected. Suppose that *word1*, *word2*, and *word3* are extracted from the API documents (see Table 1) and are tagged by the NLP parser. Also, all words are initially classified according to the POS usages. Assume that $T(WO) = 150$ and $T(PR) = 0.95$. Through the first *W-Occurrence Filter*, *word3* is filtered out because all occurrences of the word are less than $T(WO)$. Then, *POS Ratio Filter* eliminates *word2* because the highest POS ratio does not occupy the threshold $T(PR)$. Only words that passed the two filters are stored in the Code Dictionary.

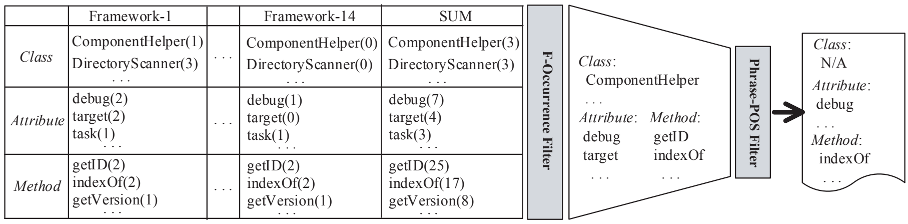


Fig. 3: Idioms identification from API Documents.

3.1.3 Extracting Idioms

In a manner similar to domain word extraction, our approach uses two subsequent filters to extract idioms. The *F-Occurrence Filter* checks if the identifier occurs in at least $T(FO_{fmw})$ different API documents, where $T(FO_{fmw})$ denotes the minimum threshold for occurrences within the frameworks. In addition, the filter includes occurrence constraints for the classes, attributes and methods, for which each threshold is indicated as $T(FO_{cls})$, $T(FO_{att})$ and $T(FO_{met})$ respectively. The second filter, *Phrase-POS filter*, figures out whether the identifier violates Java naming conventions. If it violates the conventions, the identifier is collected as an idiom into the Code Dictionary.

Figure 3 shows an example for extracting idioms from the API documents in Table 1, where $T(FO_{fmw}) = 2, T(FO_{cls}) = 3, T(FO_{att}) = 4$ and $T(FO_{met}) = 15$. The *DirectoryScanner* class identifier cannot pass the *F-Occurrence Filter* because it is only discovered in *Framework-1* even though its occurrence is over the $T(FO_{cls})$ threshold. While the *ComponentHelper* class identifier can pass the *F-Occurrence Filter*, it cannot pass the *Phrase-POS Filter* because it does not violate Java naming convention for the class identifier. The *debug* attribute, for example, passes all filters and can be identified as an idiom due to its occurrence throughout the frameworks and violation of Java naming convention for attributes. In terms of the methods, *indexOf()* is identified as an idiom because it is discovered in more than two frameworks with over 15 instances, and it violates the Java naming conventions. In reality, the method is commonly used and accepted in Java programs even though it is a violation of naming conventions. Through the preliminary study, we defined the thresholds for the evaluation.

3.1.4 Collecting Abbreviations

In addition to domain words and idiom identifiers, developers tend to use abbreviations instead fully writing out long words. For example, they normally use ‘spec’, ‘alloc’, and ‘doc’, instead of ‘specification’, ’allocation’ and ‘document’. However, most NLP parsers cannot recognize whether words are abbreviations or not. This decreases the precision of the parsing results.

Table 2: List of Abbreviated Words and Examples

Abbreviated Word	Full Word	Example	Framework
val	value	*get_val()*, *insert_val()*	JDK
dir	directory	*getSrcDir()*, *getBaseDir()*	Ant
calc	calculate	*calcSize()*	POI
concat	concatenate	*concatSystemClassPath()*	Ant
del	delete	*delFile()*	Ant
exec	execute	*exec()* *execSQL()*	JDK Ant
gen	generate	*genTile()* *genKeyPair()*	QuaQua JDK
init	initialize	*initAll()* *initCause()*	POI JDK
lib	library	*addLib()*, *exitAntLib()*	Ant
inc	increase	*incValue()*	StanfordParser
spec	specification	*getKeySpec()*, *getParameterSpec()*	JDK
alloc	allocate	*allocShapeId()*	POI
doc	document	*getDoc()* *loadDoc()*	JDK POI

To assist the NLP parsers, our approach takes a mapping from abbreviated identifiers to the original words. We initially identified all words not discovered in WordNet, and then we eliminated acronyms such as HTTP, FTP and XML because these are intuitive for developers. We then parsed these as nouns by the NLP parser. After that, we obtained the 13 abbreviations, and identified the full word for each abbreviation as shown in Table 2. These words were validated by the subjects during the manual evaluation. Although there are several techniques [8, 16] that can recover the original words from abbreviated code identifiers, our approach uses a manual approach since it is simple and effective for our purpose. Automatic abbreviation recovery remains a task for future work.

3.2 Phase 2: Detecting Inconsistent Identifiers

This section describes the second phase of our approach, which is how to detect inconsistent identifiers by using the Code Dictionary. In this phase, our approach first scans the identifiers in the source code and figures out the POS of each of the words in an identifier. Then, it detects inconsistencies based on the Code Dictionary and detection rules defined in Section 2.2.

3.2.1 POS Tagging

Similar to that of the first phase, our approach uses an NLP parser to figure out the POS of each word in the identifiers. One difference is that the identifiers are collected from the target program instead of the API documents. In addition, all abbreviated words are replaced by the full original words according to the mapping in the Code Dictionary.

3.2.2 Detecting Semantic Inconsistency

Semantic inconsistencies occur when more than two different words have a similar meaning and are both used as the same POS. This leads to confusion when reading a program, and such cases have been formally defined in Definition 6.

To detect semantic inconsistencies, our approach uses WordNet [18] to first collect semantically similar words for a given word used in an identifier. WordNet is a lexical dictionary that provides root words, senses, and synonyms of a word. This is widely used when identifying relationships between words [10, 29].

Algorithm 1 describes how our approach identifies semantically similar words. For each POS usage of a word w (Line 2), WordNet gives a set of synonyms for the given word when used as a POS (Line 3). If the synonym is observed within the target program (Line 4), similarity and reverse similarity are computed (Line 5 and 6). This similarity value represents how much the synonym syn_k is tightly coupled with the word w when used as a given POS pos_i. The reverse similarity is defined in the opposite manner. Then, if both similarity values are larger than the similarity threshold $T(SEM)$[20] predefined by the user (Line 7), the synonyms are collected as semantically similar words (Line 8).

For example, suppose that there are the verb 'get', 'acquire', and 'grow' and these are used in the target program. The verb 'get' has 30 senses. Among them, 'acquire' is the first synonym meaning, which means to 'come into the possession of something concrete or abstract', and 'grow' is the 12th synonym meaning 'come to have or undergo a change of physical features'. According to the algorithm, the semantic similarity of 'acquire' and 'grow' to 'get' is $1-(1/30) = 0.96$ and $1-(12/30) = 0.6$ respectively. Therefore, we only consider 'acquire' to be a synonym of 'get'.

Among the semantically similar words, the most frequently used word is considered to be the base word, and the others are regarded as semantically inconsistent words. Then, the identifiers containing the inconsistent words are detected as semantic inconsistencies.

Note that several previous techniques [1, 2] have tried to search for synonyms without considering the POS from WordNet. However, those are not successful since their results include too many unreliable and irrelevant synonyms. On the other hand, our approach uses a word and its POS together to search for more reliable and relevant synonyms, improving the accuracy of inconsistency detection.

3.2.3 Detecting Syntactic Inconsistency

Syntactically inconsistent identifiers have similar letter sequences, as defined in Definition 7. These identifiers often cause confusion when trying to understand the source code of a program. In addition, they can lead to incorrect refactorings since some refactorings are sensitive to spelling.

[20] Decision of this threshold is carried out in the preliminary study.

Algorithm 1: Collecting semantic similar words.

Input : W: a set of words used in the project.
Input : w: target word ($w \in W$).
Input : STR: similarity threshold.
Input : $POS(w)$: a set of POSes (only observed in the project) for w.
Input : $syn(w, pos_i)$: a set of synonyms for w when used as pos_i. This set is provided by WordNet [18].
Input : $synidx(syn_k, w, pos_i)$: the rank of syn_k in the sense list of w when used as pos_i. The rank is provided by WordNet.

Output: SSW: a set of semantically similar words.

let $SSW \Leftarrow \emptyset$;
foreach $pos_i \in POS(w)$ **do**
  foreach $syn_k \in syn(w, pos_i)$ **do**
    if $syn_k \subset W$ **then**
      $semsim \Leftarrow 1 - synidx(syn_k, w, pos_i)/|syn(w, pos_i)|$;
      $rsemsim \Leftarrow 1 - synidx(w, syn_k, pos_i)/|syn(syn_k, pos_i)|$;
      if $semsim > STR$ *and* $rsemsim > STR$ **then**
        let $SSW \Leftarrow syn_k$;
      end
    end
  end
end

To detect syntactically similar identifiers, our approach first identifies the syntactically similar words. The syntactical similarity is dependent on the edit distance of two words. The approach leverages the Levenshtein Distance Algorithm [30] used to compute $DIST(w_1, w_2)$ in Definition 7. This algorithm measures the distance between two words by counting the alphabetic differences and dividing them with the number of letters. For example, the distance between *kitten* and *sitting* is three (*kitten* $\rightarrow$ ***s****itten* $\rightarrow$ *sitt****i****n* $\rightarrow$ *sittin****g***).

This approach used the distance to compute a determinant value, $C(w_1, w_2)$ in Definition 7. If the value is larger than the threshold $T(SYN)$, the two words are considered to be syntactically similar. For the above example where $T(SYN) = 4$, $C(kitten, sitting) = \frac{7}{(|6-7|+1)\cdot 3} = 1.17$. Thus, *kitten* and *sitting* are not syntactically similar. On the other hand, *credential* and *credental* shown in Section 2.2 are syntactically similar since $C(credential, credental)) = \frac{10}{(|10-9|+1)\cdot 1} = 5$. The threshold $T(SYN)$ is computed in the evaluation section.

Our approach takes an arbitrary pair of identifiers and determines whether the identifiers have syntactic inconsistencies. Once the two identifiers include any pair of syntactically similar words, they are syntactically inconsistent according to Definition 7. The approach designates the identifier less frequently used in the target program as an inconsistent identifier.

One exception is for noun words that have the same root word in WordNet [18] because these are often used on purpose. For example, '*String accent*' and '*String[] accents*' are syntactically similar words. However, a developer can use a plural form on purpose for this case.

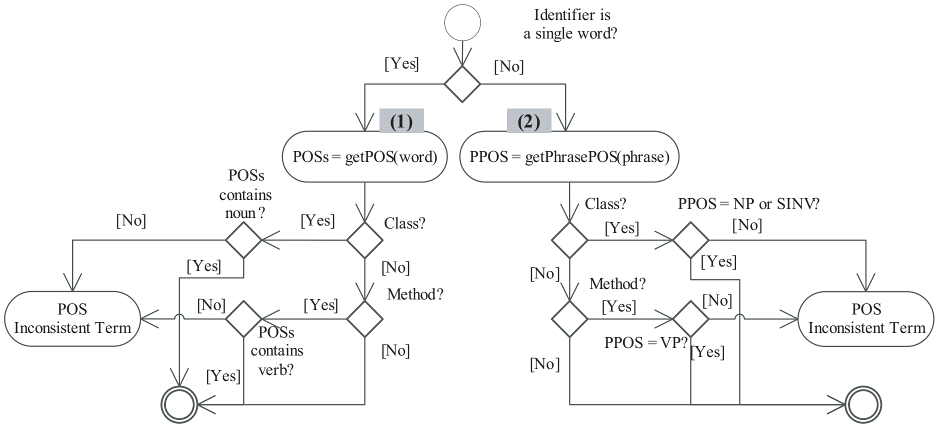


Fig. 4: POS-inconsistent identifiers detection.

This exception can improve the accuracy of our approach since it can filter out unnecessarily detected identifiers.

3.2.4 Detecting POS Inconsistency

There are two types of POS inconsistencies: Word and Phrase-POS inconsistencies. Word-POS inconsistent identifiers have an identical word that is used with different POSes as defined in Definition 8. For phrase-POS inconsistency, our approach detects the identifiers that violate the grammatical rules of Java naming conventions according to Definition 10.

In particular, for word inconsistency, only less frequent POSes are regarded to be inconsistent. The identifiers that contain the dominant POS words are not detected as inconsistencies. For example, when 90% of the instances of *abort* use it as a verb, the remaining cases are the inconsistent ones.

Phrase-POS inconsistent identifiers are detected by comparing the POS of an identifier to the grammatical rules of the Java naming conventions. Figure 4 shows an algorithm that can be used to detect phrase-POS inconsistent identifiers. There are two cases where the number of words consisting of an identifier should be considered. If the target identifier is a single word (see the flow (1)), our approach checks if a specific POS exists in WordNet. For example, if a single word is used as a class identifier and the word can be a *noun* according to WordNet, then it is considered to ba a valid identifier. If not, the approach detects the identifier as inconsistent.

Our proposed approach first parses composite identifiers by using an NLP parser to get their phrase-POS (PPOS). Then, it detects inconsistency by comparing the *PPOS* to the grammatical rules (see the flow (2)). For example, if the class identifier is not a noun phrase, it violates naming conventions, which leads to a phrase-POS inconsistency.

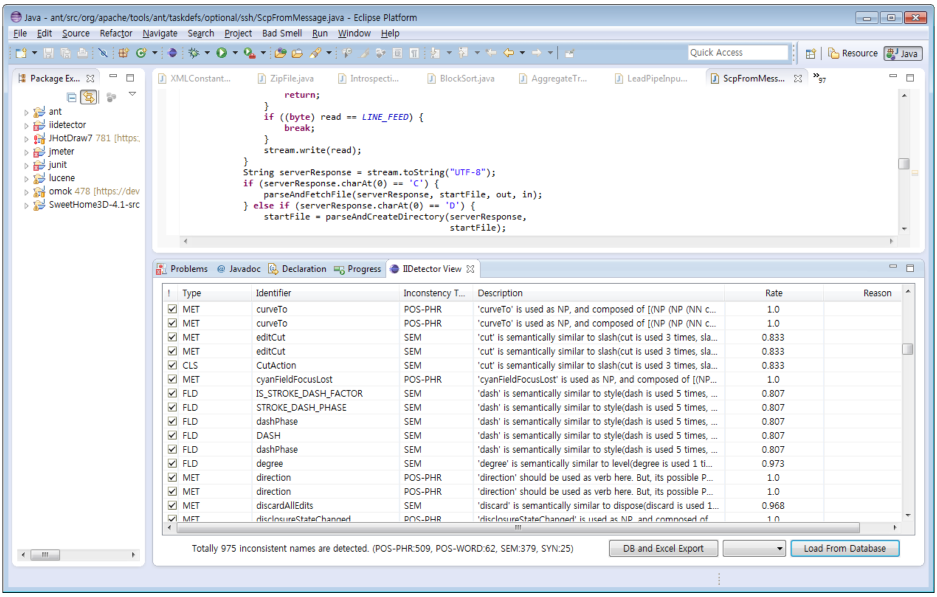


Fig. 5: Snapshot of CodeAmigo.

3.2.5 Filtering and User Feedback

After detecting the three types of inconsistent identifiers, our approach filters out inconsistent identifiers that contain domain words and idioms recorded in the Code Dictionary in order to reduce false alarms. For domain words, our approach checks whether an identifier has a domain word with the corresponding POS specified in the Code Dictionary.

A user of our method can provide feedback if any detection is incorrect. In case of semantic or syntactic inconsistency, the user can suggest exceptions to the rules in order to accept the detected inconsistency. For POS inconsistency, POS rules can be updated, or specific identifiers can be ignored.

3.3 CodeAmigo: Tool Support

We developed an Eclipse based tool named *CodeAmigo* [13]. It provides a graphical interface for developers to be able to use our approach easily. The tool takes a project and scans all of the source code files in the project in order to generate a report, as shown in Figure 5. This report lists all of the inconsistent identifiers that were detected by our approach and describes the potential causes.

4 Evaluation

This section describes the results from an experiment that was designed to evaluate our approach, as presented in Section 3. This experiment consists of a preliminary study and the subsequent inconsistent identifier detection. The preliminary study was conducted to find out appropriate threshold values for our approach. We then performed the second experiment using these threshold values. This experiment uses our approach to detect inconsistent identifiers.

We first collected seven popular software projects written in Java. Six of these were open source projects and one was our tool support project *CodeAmigo*, as shown in Table 7. Apache Lucene is an open source project that can be used to build a search engine. Apache Ant, Apache JMeter and JUnit are support tools used to build Java-based applications. JHotDraw and Sweet Home 3D are GUI-based tools used to support graphic editing and virtual furniture placement, respectively.

To check the validity of the inconsistent identifiers detected by our approach, we asked 16 developer and measured the precision and recalls [31]. In addition, we conducted an interview for six of these participants in order to find out the effectiveness of our approach. The remainder of this section shows the results for the inconsistent identifiers that were detected by our approach in Section 4.2 and further presents quantitative and qualitative results obtained from the experiment in Section 4.3.

4.1 Preliminary Study: Deciding Threshold Values

To effectively build a Code Dictionary and detect inconsistent identifiers, appropriate threshold values should be identified. Thus, we first conducted a sensitivity analysis for the threshold values defined in Section 3.

4.1.1 Threshold values for a Code Dictionary

We first examined two threshold values for $T(WO)$ and $T(PR)$, which are necessary to identify domain words. As described in Section 3.1.2, these values are used in the *W-Occurrence Filter* and *POS-Ratio Filter*, respectively. To figure out appropriate thresholds, we varied these two values as independent variables and applied the values to the training set listed in Table 1. For $T(WO)$, we used three different values: 80, 100, and 120 while $T(PR)$ was varied by 0.8, 0.9, and 0.95. Then, we manually checked whether the domain words are correctly identified for each combination of two thresholds. The results are shown in Table 3. *#Detection* in Table 3 represents the number of detected domain words after applying two filters shown in Figure 2 with threshold values of $T(WO)$ and $T(PR)$. *Precision* is the ratio of true positive domain words out of all detected words after manual checking.

Since it is necessary to consider $T(WO)$ and $T(PR)$ together to find appropriate thresholds of the *W-Occurrence Filter* and *POS-Ratio Filter*, we defined Equation 1 as a selection

Table 3: Sensitivity analysis results for different threshold values to detect domain word POS. The first two columns show different values for $T(WO)$ and $T(PR)$ as independent variables. The next three columns are the results (dependent variables) when applying each combination of two thresholds to the training set listed in Table 1. *#Detection* is the number of detected domain words with respect to each combination of $T(WO)$ and $T(PR)$ while *Precision* is the ratio of true positive domain words after manual checking. The underlined numbers are the maximum values for each column while the wavy underlined are the minimum values. After applying Equation 1, $T(WO) = 100$ and $T(PR) = 0.8$ are selected as threshold values for the main experiments described in Section 4.2.

$T(WO)$	$T(PR)$	#Detection	Precision	Selection F.
80	0.8	238	87.4%	0.0
80	0.9	223	88.8%	14.0
80	0.95	189	90.5%	20.1
100	**0.8**	191	92.1%	31.9
100	0.9	179	92.7%	29.3
100	0.95	152	94.1%	18.0
120	0.8	152	95.4%	21.5
120	0.9	144	95.8%	15.7
120	0.95	126	96.0%	0.0

Table 4: Sensitivity analysis results for different threshold values used to identify idioms.

		$T(FO_{met})$					
$T(FO_{fmw}) = 2$		5		10		15	
$T(FO_{cls}) = 2$		Det.	Pre.	Det.	Pre.	Det.	Pre.
$T(FO_{att})$	2	130	87.7%	**88**	**93.2%**	61	96.7%
	3	122	86.9%	80	92.5%	53	96.2%
	4	120	86.7%	78	92.3%	51	96.1%

factor. This equation computes an incorporated value based on the number of detected domain words and its precision. In this equation, $min(*)$ and $max(*)$ indicate that the minimum and maximum values of *#Detection* and *Precision* columns in Table 3 ($col[*]$ is a set of values in a specific column in Table 3). Using this equation, we selected $T(WO) = 100$ and $T(PR) = 0.8$, respectively, as their selection factor was the highest value (=31.9).

$$Selection.F = \frac{Precision - min(col[Precision])}{max(col[Precision]) - min(col[Precision])} \times \frac{\#Detection - min(col[\#Detection])}{max(col[\#Detection]) - min(col[\#Detection])} \tag{1}$$

We carried out another sensitivity analysis of threshold values for extracting idioms described in Section 3.1.3. Similar to the above analysis, we examined several combinations of $T(FO_{fmw}), T(FO_{cls}), T(FO_{att})$ and $T(FO_{met})$. Basically, every threshold values must be

Table 5: Sensitivity analysis results for different threshold values to detect semantic inconsistency.

Ant	$T(DOM)$					
	0.8		0.9		0.95	
$T(SEM)$	Det.	Pre.	Det.	Pre.	Det.	Pre.
0.8	514	70.2%	223	79.1%	124	73.8%
0.9	401	67.7%	**161**	**83.9%**	84	79.3%
0.95	378	75.8%	113	83.3%	46	69.6%

larger than one because the a single identifier must be discovered at least two times to decide if it is an idiom within the frameworks, classes, attributes, and methods. We counted the number of idiom detections and computed their precision values by manually examining the correctness of the detections. The results are shown in Table 4.

We observed that $T(FO_{cls})$ does not affect to the results in the case of $T(FO_{fmw}) \geq 2$, implying that any class identifiers do not have the same name throughout all of the frameworks listed in Table 1. In addition, In case of $T(FO_{fmw}) \geq 3$, $T(FO_{att})$ cannot affect the results. Thus, $T(FO_{fmw})$ was set to 2 in order to have $T(FO_{att})$ affect the idiom detection. Higher values of $T(FO_{met})$ can increase the precision. The decrease in precision for $T(FO_{met})$, in accordance with $T(FO_{att})$, is caused by filtering out the correct idioms due to $T(FO_{att})$. This is attributed to the precision of the NLP parser. According to Equation 1, we obtained the following threshold values: $T(FO_{fmw}) = 2$, $T(FO_{cls}) = 2$, $T(FO_{att}) = 3$, and $T(FO_{met}) = 10$.

4.1.2 Deciding threshold values for inconsistent identifier detection

The threshold values used to detect inconsistent identifiers, which are described in Section 3, include $T(SEM)$ for deciding semantically similar words; $T(SYN)$ for deciding syntactically similar words; and $T(DOM)$ for a base word. $T(DOM)$ is intended for use in determining the dominant word for searching non-dominant words used as inconsistent identifiers. In order to find the appropriate threshold values, we preliminarily detected the inconsistent identifiers within *Apache Ant* and *Apache Lucene* by controlling the threshold values. We have obtained results as shown in Table 5, and the results for *Apache Lucene* that are similar to that of *Apache Ant* are omitted. Based on Equation 1, we decided that $T(SEM)$ and $T(DOM)$ should be both 0.9. For $T(SYN)$, we decided as $T(SYN) = 3$ because our approach may not effectively detect syntactically inconsistent identifiers when $T(SYN) >= 3.5$, as shown in Table 6.

4.2 Inconsistent Identifiers Detected by Our Approach

Table 8 shows the result of the inconsistency detection by our approach, where *POS-PHR*, *POS-WORD*, *SEM* and *SYN* represent phrase-POS, word-POS, semantic, and syntactic in-

Table 6: Sensitivity analysis results for different threshold values to detect syntactic inconsistency.

	$T(DOM)$			
	0.8		0.9	
$T(SYN)$	Detection	Precision	Detection	Precision
3	**10**	**80.0%**	6	66.6%
3.5	6	66.6%	1	100%
4	6	66.6%	1	100%
5	3	100%	1	100%

Table 7: The subjects used in the evaluation. CLS, MET, and ATTR represent the number of class, method, and attribute identifiers, respectively.

Subject	Version	SLOC	CLS	MET	ATTR	All
Apache Lucene [32]	3.0.3	104,287	1,279	10,218	5,526	17,023
Apache Ant [33]	3.0.3	45,835	659	4,550	2,288	7,497
Apache JMeter [34]	2.9	90,714	1,104	8,710	5,346	15,160
JUnit [35]	4.0	6,588	186	986	205	1,337
JHotDraw [36]	7.0.6	32,179	405	3,620	894	4,919
Sweet Home 3D [37]	4.10	82,439	618	4,933	3,485	9,036
CodeAmigo	1.0	6,191	46	348	160	554
Total		368,233	4,297	33,365	17,904	55,526

Table 8: The number of inconsistent identifiers detected by our approach.

Subject	POS-PHR	POS-WORD	SEM	SYN	# of detected inconsistencies	# of detected identifiers	% of inconsistent identifiers
Lucene	665	83	32	10	790	358	2.10%
Ant	599	108	135	6	848	483	6.44%
JMeter	528	80	180	13	801	379	2.50%
JUnit	148	6	35	0	189	123	9.20%
JHotDraw	509	62	130	25	726	323	6.57%
S.Home3D	238	118	77	5	438	256	2.83%
CodeAmigo	30	1	3	0	34	30	5.42%
Total	2,717	461	592	59	3,826	1,952	3.52%
%	71.0%	12.0%	15.5%	1.5%	100%		

consistent identifiers, respectively. *Total* indicates the total number of inconsistent identifiers of each project and type. Note that *# of detected identifiers* is the unique number of identifiers detected as inconsistencies and one single identifier can have several different inconsistencies. The *% of inconsistent identifiers* is the ratio of the inconsistent identifiers.

The phrase-POS inconsistency accounts for 71% of the total number of inconsistent identifiers while word-POS, semantic, and syntactic inconsistencies account for 12%, 15.5%, and 1.5%, respectively. This implies that Phrase-POS is the most frequently occurring inconsistency where identifiers violate the grammatical rules of Java naming conventions. Inconsistent POS usage (POS-WORD) and synonyms (SEM) are frequent inconsistency types as well, and syntactically inconsistent identifiers (SYN) accounts for the least number of inconsistencies.

More inconsistent identifiers are detected for Ant, JUnit and JHotDraw than other subjects. This is due to those subjects using more noun phrases instead of verb phrases for their method identifiers (e.g., *componentToRGB*).

The Code Dictionary plays a role in filtering idioms and violations of the POS of domains at the end of detection of the inconsistent identifiers. Such a process can eliminate false alarms and can improve the precision of the detections. Figure 6 shows the intermediate states of the detections for *Ant*, *Lucene* and *JMeter*. The idioms from the Code Dictionary are used as an *Idiom Filter* in order to eliminate the detected inconsistent identifiers when the identifier is discovered to be an idiom (see Table 14). For example, the method identifier *intValue()* is detected as a POS-PHR inconsistency because the method identifier should be composed of a verb or a verb phrase. However, since *intValue()* is an idiom, it should not be detected as an inconsistent identifier. Thus, it is filtered out by the *Idiom Filter*. After the *Idiom Filter* is run, the *Domain Word POS Filter* checks to see if the words and their POS that caused a detection exist in the Domain Word POS, as shown in Table 13. This filter has an effect that reduces the invalid NLP parsing. For example, the word *path* in the method identifier *mapPath()* of the Ant project is parsed as a *verb*. However, it is invalid parsing. Since the word *path* is stored in the Domain Word POS as a *noun*, the detection is invalid, and then it should be filtered out. In particular, the *Domain Word POS* filter is effective in reducing false alarms for the *POS-WORD* and *SEM*.

The following present actual examples of inconsistent identifiers that were detected by our approach. More details are available in our project web site[21].

Phrase-POS Inconsistency:

- `outStreams` (method, Ant) - ‘outStreams’ is used as a FRAG, and is composed of [(FRAG (ADVP (RB out)) (NP (NNP Streams)) (. .))]. The methods should be named as a verb phrase.
- `readerIndex` (method, Lucene) - ‘readerIndex’ is used as an NP, and is composed of [(NP (NP (NN reader)) (NP (NNP Index)) (. .))]. The methods should be named as a verb phrase.
- `Fail` (class, JUnit) - ‘Fail’ should be used as a noun here. But its possible POSes include [verb].

Word-POS Inconsistency:

[21] https://sites.google.com/site/detectinginconsistency/

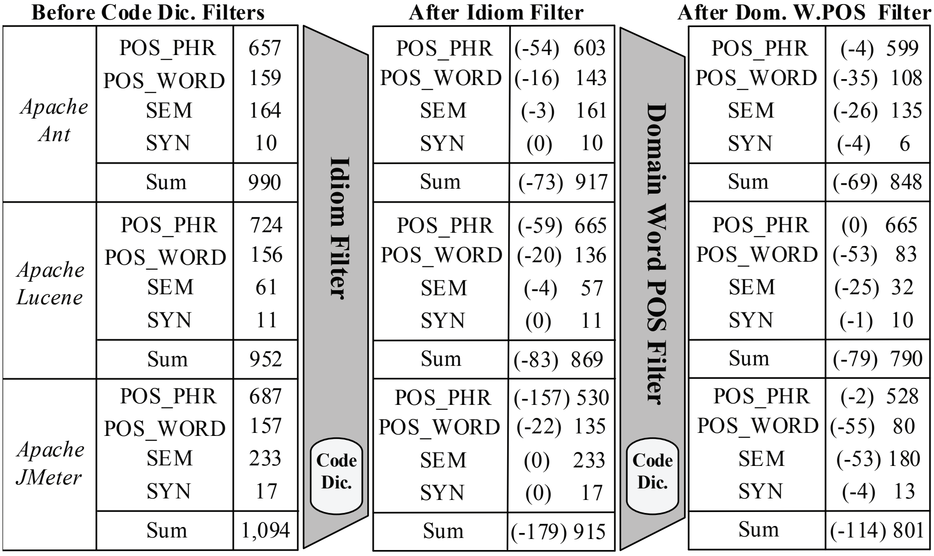


Fig. 6: Filtering intermediate inconsistent identifiers by using the code dictionary.

- *outStreams* (method, Ant) - 'out' is generally used as a noun (120/123, 0.975), but here it is used as an adverb (1/123, 0.008).
- *CCMCreateTask* (class, Ant) - 'create' is generally used as a verb (341/357, 0.955), but here it is used as a noun (16/357, 0.044).
- *DrawApplet* (class, JHotDraw) - 'draw' is generally used as a verb (148/155, 0.954), but here it is used as a noun (7/155, 0.045).

Semantic Inconsistency:

- `Specification` and *spec (Ant)* - 'spec' is semantically similar to specification (spec is used 7 times, specification is used 37 times).
- `Selector` and *Chooser (Sweet Home3D)* - 'selector' is semantically similar to chooser(selector is used 1 times, chooser is used 8 times).
- `fetch` and *get (JMeter)* - 'fetch' is semantically similar to get (fetch is used 2 times; get is used 2224 times).

Syntactic Inconsistency:

- `startsWith()` (method, JMeter) - 'starts' is syntactically similar to start(starts is used 1 times; start is used 45 times)".
- `getPreserve0Permissions() (method, Ant)` - 'preserve0' is sytactically similar to preserve (preserve0 is used 1 times; preserve is used 14 times).

4.3 Analysis of Inconsistency Detection

To validate our approach, we establish four research questions (RQs) as follows:

1. *RQ1: How precise are the inconsistencies detected by our approach?*
2. *RQ2: How comprehensive are the detection results?*
3. *RQ3: How much does Code Dictionary contribute to reduce false positives?*
4. *RQ4: How useful are the detection results for developers?*

The followings describe the experiment setting and the analysis of the results.

4.3.1 RQ1: How precise are the inconsistencies detected by our approach?

Although our approach attempted to thoroughly detect identifier inconsistencies, as described in Section 3, the detection results can be subjective since different developers may have a different sense of inconsistency. This subjectivity is a common issue in NLP-related work [38]. In addition, the parser [22] used in our approach is intended to parse natural language instead of source code identifiers, even though the parser has a high precision[22]. Hence, it is necessary to evaluate the results of the detection with human subjects.

To answer RQ1, we gathered 16 volunteer practitioners with 3 to 15 years of development experience as human subjects for our experiment. They are currently mainly developing Java-based software systems such as package solutions and enterprise applications, as shown in Table 9. We developed a web-based system [39] that presents all our detection results for the seven projects, and stores all subjects' validation results to facilitate this experiment. This system provides the name, type, and reason of each inconsistent identifier. We had a three-hour workshop to distribute CodeAmigo with the seven projects, introduce each project and their major features, and explain the web-based system for evaluation. Then, we asked the subjects whether the detection results by our approach were correct or not. During the workshop, the subjects validated inconsistent identifiers and checked the actual source code where the inconsistent identifiers were used. After the workshop, we requested the subjects to complete the evaluation within a week. During the one-week evaluation, all subjects could access the source code of each of the detection results whenever they wanted to see the contextual information such as the parameters of the method identifiers and type information for the field identifiers.

In order to evaluate the validity of our approach, we applied a traditional precision and recall measure [31] instead of measures such as the area-under-ROC curve [40] with the following reasons. First, it is almost impossible to manually find true-negative identifiers for the 7 projects' source code. Second, we have defined the thresholds in the preliminary study, and changing the thresholds means that all manual evaluation processes should be conducted from the start to obtain a new confusion matrix. Third, we considered that the precision and recall measure is sufficient to explain the efficiency of our approach, since true-negative detections are not considered for the precision measure.

[22] The Stanford parser [22] has 86% parsing precision for a sentence consisting of 40 English words.

Table 9: Work experience and expertise of human subjects.

Work Experiences		Expertise	
1 – 5 years	6	package solutions	6
5 – 10 years	3	enterprise applications	3
11 – 15 years	7	mobile applications	5
Total	16	middlewares	2

The results obtained from the human subjects were used to measure the precision of the detection results of our approach according to the following equation [31]:

$$\text{Precision} = \frac{|(H_{id} \cap D_{id})|}{|D_{id}|} \tag{2}$$

where D_{id} is the set of inconsistent identifiers detected through our approach and H_{id} is the set of identifiers marked as a correct detection by the human subjects. Note that $(H_{id} \cap D_{id}) = H_{id}$ since H_{id} is a proper subset of D_{id}.

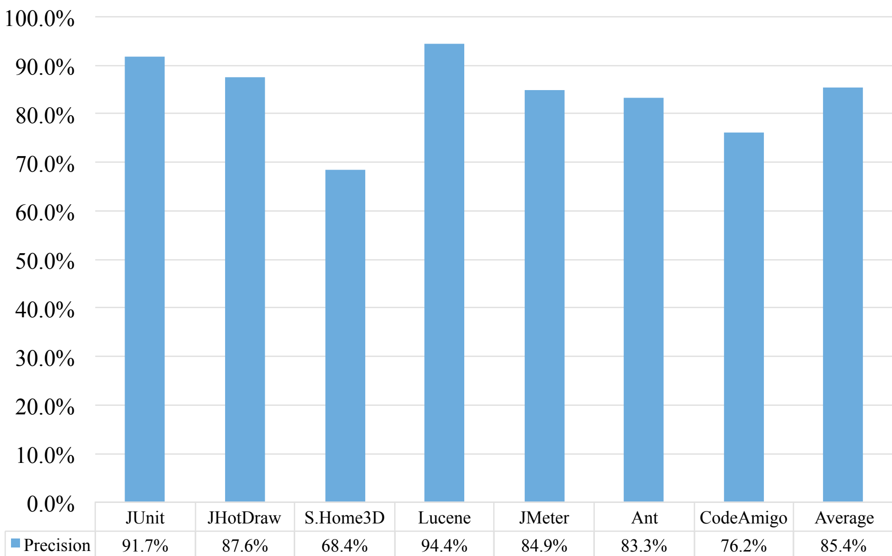


Fig. 7: Precision result for each project shown in Table 7. These results are calculated by Equation 2.

Figure 7 shows the precision results for the projects. The average precision for the seven projects is of 85.4% with a minimum precision of 68.4% for "*Sweet Home3D*" and a max-

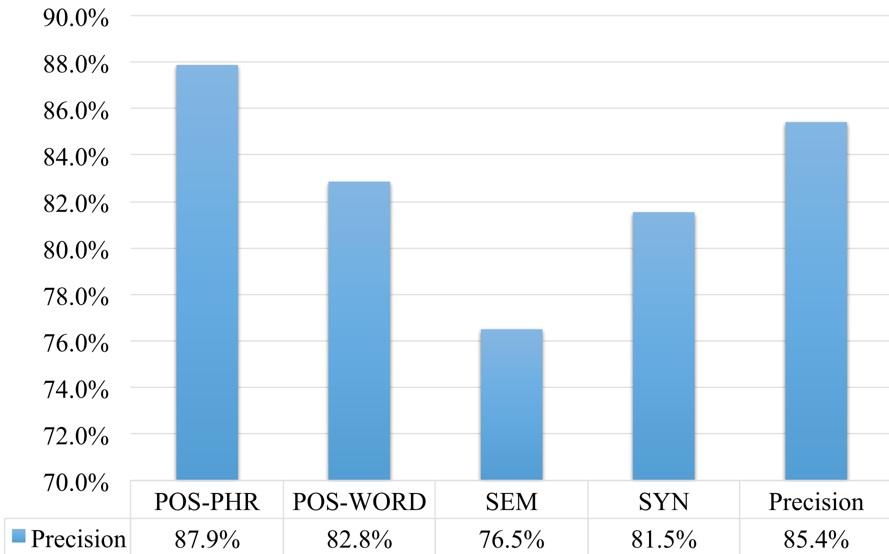


Fig. 8: Precision of each inconsistency types.

imum precision of 94.4% for "*Lucene*". This implies that at least 8 out of the 10 identifiers are actually inconsistent once they are detected by our approach.

Among the subjects, Lucene showed the highest precision. This is supposedly possible since it has many identifiers composed of natural words (e.g., *documents* and *parse*), and even the technical terms in the project are quite similar to those of natural language. This could increase the precision of the NLP parsing.

On the other hand, Sweet Home3D showed the lowest precision. One of the main reasons for its low precision is that the NLP parser did not correctly tag the POS of the most common words, such as *furniture*, *plan*, and *piece*, in the project.

In addition, JHotDraw, JMeter and Ant were less precise compared to Lucene because they include diverse terms for GUI specific words such as *Panel* and *Frame*, which can often result in incorrect POS tagging.

When it comes to inspecting the results of each inconsistency type, the phrase POS inconsistency (POS-PHR) has the highest precision (87.9%), as shown in Figure 8. This indicates that our approach can detect naming rule violations with precision. On the other hand, semantic inconsistencies (SEM) have the lowest precision at 76.5%. This low precision indicates that the human subjects often regarded for the words that were detected semantically similar were not exactly the same. However, this is still positive since it implies that approximately 7 out of 10 semantically inconsistent identifiers detected through our approach were correct.

Additionally, we examined the correlation between development experience and precision. By using linear regression, we computed correlation of all 16 developers with respect to the precision value of each developer. As a result, the correlation value was -0.419. However, this is not conclusive since R^2 was 0.175. Therefore, there was no significant correlation between them.

4.3.2 RQ2: How complete are our detections?

In addition to the precision, it is important to find out whether our approach can completely detected inconsistent identifiers in a program with few number of missing inconsistencies (i.e., recall [31]). Note that the entire set of inconsistent identifiers is necessary in order to compute a recall value. However, it is difficult to find every inconsistent identifier in the project by employing human developers. Thus, we observed to what extent our approach could detect inconsistent identifiers that were missed by human subjects instead of computing the traditional recall value.

We conducted two experiments where the subjects 1) manually detected the inconsistent identifiers from scratch and 2) where they manually detected them with the assistance of our tool. Six subjects participated in this set of experiments, and they had 10–15 years of experience in Java development. We ran a small workshop to introduce them to our experiment, and we distributed work-sheets containing all of the identifiers and their types (e.g., class or method), all words, and all identifiers that include each word. The reason for which we provided additional materials for the experiment is that manually collecting all cases of word usages throughout the project is tedious and time-consuming work. In addition, we provided Eclipse with the seven projects without showing CodeAmigo. The experiment was conducted for three hours, and after the experiment, we carried out a semi-structured interview. We selected the JUnit as an experiment project because the number of its identifiers is relatively small compared to other projects listed in Table 7. The JUnit project contains the 941 identifiers and the 665 unique words consisting of the identifiers.

For the first experiment, we showed the source code of JUnit and designated all identifiers in the code by highlighting their type (e.g., class, method, and attribute). The participants examined the identifiers and marked whether or not they were inconsistent. The objective of this experiment was to figure out how many inconsistent identifiers the participants could detect manually. This may reflect how effectively developers can detect inconsistency during real development.

The second experiment was conducted to observe how well our approach could enhance inconsistency in the detection. We provided the detection results of our approach after the first experiment. Then, we asked the participants to check the validity of their detection results in the first experiment and of any missing identifier as well.

Equation 3 shows how we computed the recall of our approach. D_{id} is the set of identifiers that were detected by our approach, and M_{id} is the set of identifiers that are manually detected by the participants in the above experiments. In addition, we computed the F-measure by using Equation 4.

$$\text{Recall} = \frac{|(D_{id} \cap M_{id})|}{|M_{id}|} \quad (3)$$

$$\text{F-measure} = \frac{2 \cdot Precision \cdot Recall}{Precision + Recall} \quad (4)$$

Figure 9 shows the results of the above mentioned experiment. For this figure, we computed the average of each measure for every individual participant. In the case of the purely manual detection, the recall, precision, and the F-measure values were 55.80%, 64.43%, and 59.80%, respectively. Note that our approach detected 123 inconsistent identifiers (D_{id}) as shown in Table 8. The participants detected 143.25 inconsistent identifiers (M_{id}) and $|(D_{id} \cap M_{id})|$ was 79.25 on average.

After providing the detection result of our approach, we observed how the participants changed their detection results. As a result, they added more inconsistent identifiers when they accepted detection results of our approach, which is shown in the second row of Figure 9. They detected 24 (16.7%) additional inconsistent identifiers on average, which leads to a 30% increment in $|(D_{id} \cap M_{id})|$. Based on this result, we observed an improvement for each measure; 6.34% for recall, 19.51% for precision, and 11.34% for F-measure, respectively.

	Recall	Precision	F-Measure	Detections
Manual	55.80%	64.43%	59.80%	D 123 (79.25) 143.25 M
Manual + Our Approach	62.14%	83.94%	71.14%	D 123 (103.25) 167.25 M
Improvement	6.34%	19.51%	11.34%	24(30%) 24(16.7%)

Fig. 9: Recall and F-Measure for pure manual detection and manual detection supported by our approach.

	Recall	Precision	F-Measure	Detections
Manual (Union)	48.07%	91.06%	62.92%	D 123 (112) 233 M
Manual (Union) + Our Approach	48.07%	91.06%	62.92%	D 123 (112) 233 M
Manual (Intersection)	79.61%	66.67%	72.57%	D 123 (82) 103 M
Manual (Intersection) + Our Approach	83.59%	86.99%	85.26%	D 123 (107) 128 M

Fig. 10: Recall and F-Measure for intersection/union set of manual detection.

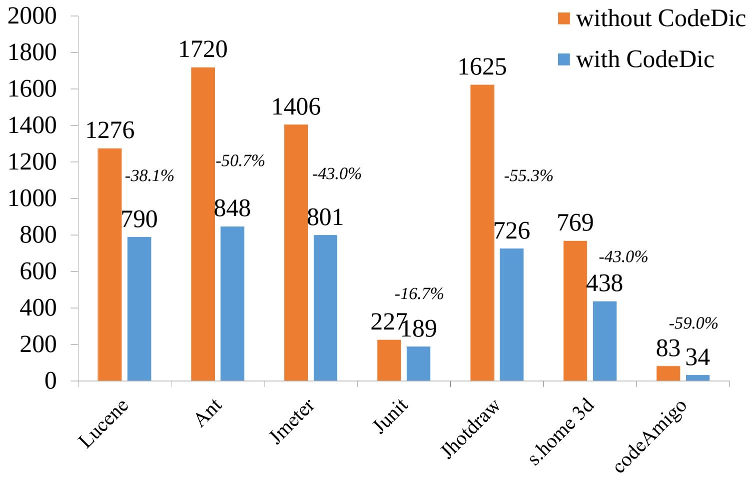


Fig. 11: Code Dictionary's contribution for reducing false positives.

We conducted an additional analysis to simulate a real development environment since *inconsistency* can be subjective for each of the human subjects. Developers normally have a formal/informal meeting to discuss the validity of their detections. Thus, we further investigated their detections with respect to two cases. First, if we assumed that all detections for every participant were correct, then they detected 233, as shown in Figure 10 (Union), and the results were 48.07% for recall and 91.06% for precision. In this case, our approach could not improve their detections.

On the other hand, if we assumed that the correct detections are just those where all participants commonly agreed, then the recall and precision of our approach were 79.61% and 66.67%, respectively, as shown in Figure 10 (Intersection). In addition, the values improved to 83.59% for recall and 86.55% for precision when our approach helped them during the tasks. Note that our approach could give advices on 25 more inconsistent identifiers (from 82 to 107).

4.3.3 RQ3: How much does Code Dictionary contribute to reduce false positives?

To answer this RQ, we compared inconsistent identifiers detected by our approach with-/without Code Dictionary. Note that the detection results with Code Dictionary is a subset of detection results without Code Dictionary since it may filter out false positives. Figure 11 shows the results; Code Dictionary reduced 43.7% of potential false positives on average.

The considerable number of inappropriate NLP parsing was filtered out by Code Dictionary containing the POS of domain words, idioms, and mapping between abbreviations and its original words. First, the POS of domain words could filter out diverse parsing errors in analyzing identifiers. The representative samples are summarized as below:

- *header* (method identifier in Ant) is parsed as '(S (VP (VB header)) (. .))', which is a wrong parsing result. However, it is filtered out because the word *header* is stored as a noun in Code Dictionary (see Appendix A).
- *setupPage* (method identifier in Sweet Home3D) is parsed as '(S (NP (NN setup)) (VP (VBZ Page)) (. .))'. The word *page* is stored as a noun, so that the identifier has been filtered out.
- *toImage* (method identifier in JHotDraw) is parsed as '(S (VP (TO to)(VP (VB Image)))(. .))'. The word *Image* is incorrectly parsed. However, our approach could filter out the identifier by using Code Dictionary that classifies the word *Image* as a noun.
- *offset* (attribute identifier in JMeter) is parsed as '(S (VP (VB offset)))'. Code Dictionary records the word *offset* as a noun.
- *fileSize* (method identifier in Ant) is parsed as '(S (VP (VB file)(NP (NN Size)))(. .))'. Code Dictionary classifies the word *file* as a noun in the storage.

Second, the idioms of Code Dictionary could filter out diverse false positives as below:

- *actionPerformed* (method identifier in JHotDraw and JMeter) is parsed as '(S (NP (NN action)) (VP (VBD Performed))(. .))'. However, the identifier is being commonly used in source code written in Java regardless of the right or wrong NLP parsing. Thus, it is filtered out by Code Dictionary (see Appendix A).
- *available* (method identifier in Ant and JHotDraw) is parsed as '(FRAG (ADJP (JJ available))(. .))'. The word *available* is also incorrectly parsed by the NLP parser, so that it could have been detected as an inconsistent identifier by *CodeAmigo* without Code Dictionary support. Code Dictionary could filter out the identifier because it is recorded as an idiom.
- *indexOf* (method identifier in JHotDraw) is filtered out because it is an idiom.

Third, Code Dictionary also maintains mapping between a word and its abbreviation. In some cases, replacing an abbreviation into its original word has an influence on appropriate NLP parsing. The representative cases are presented as below:

- *initSegmentName* (method identifier in Lucene) is parsed as '(S (NP (NNP init)(NNP Segment)) (VP (VB Name))(. .))'. The NLP parser analyzed *init* as a noun when *init* was not recovered into the original word *initialize*. After recovering it into the original, the identifier is parsed as '(S (VP (VB *initialize* (S (NP (NNP Segment)(VB (VB Name)))(. .)))))'. Although all words were not correctly parsed, the word *initialize* has been correctly parsed after recovering it.
- *specFile* (attribute identifier in Ant) is parsed as '(NP (JJ spec) (NNP File))'. After replacing the word *spec* into *specification*, the NLP parser correctly analyzed it as '(NP (NNP *specification*) (NNP File))'.

In addition to this, the mapping information in Code Dictionary could recover diverse identifiers such as *initLookAndFeel* (method identifier in Sweet Home3D), *dirListing* (attribute identifier in Ant), *closeDir* (method identifier in Lucene), and *checkJarSpec* (method identifier in Ant).

As a result, Code Dictionary could bridge the gap between source code analysis and natural language analysis by using domain words POS, idioms, and abbreviations generally used in writing source code. Thus, it can help inconsistency detection by reducing diverse false positives caused by NLP parsing errors.

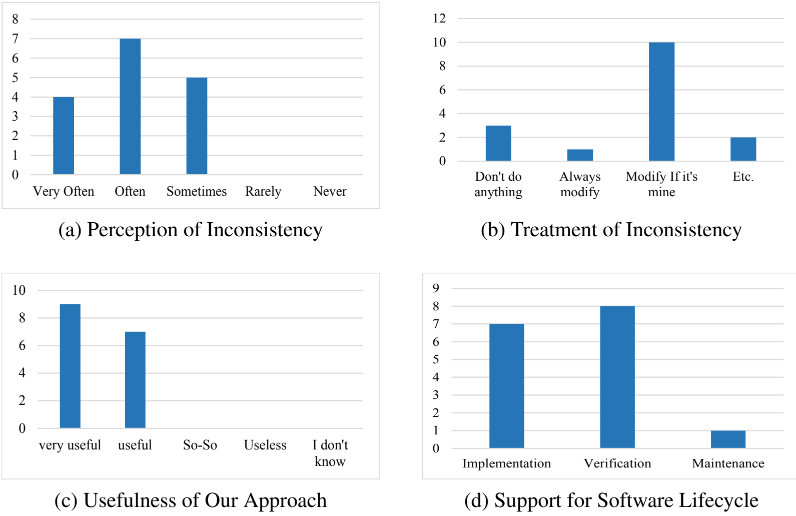
4.3.4 RQ4: How frequently do developers discover inconsistent identifiers and how useful are our detections?

We performed a semi-structured interview with questionnaires for all participants to further elaborate our findings from Section 4.3.2. This interview investigated the necessity for this approach and asked several that were prepared questions as follows:

Questions on the Necessity of the Detection of Inconsistent Identifiers

- *How often do you encounter an inconsistency of identifiers?:* Most of the participants stated that they often see inconsistent identifiers in their source code as shown in Figure 12(a). Also, they emphasized that these are more frequently discovered as the scale of the project becomes larger. The extent of the discovery is different depending on the role of the projects. When they have any responsibility to assure the quality of the source code, they more frequently discovered inconsistent identifiers.
- *What do you do when you see inconsistent identifiers?:* Most of the subjects did not correct the identifiers if they were not in charge of the source code (see Figure 12(b)). Even if they are the authors of the source code, they did not change it unless they were in charge of maintenance of the source code. If the code belongs to others, they do not modify it because they do not want to make controversial issues that could result from the modification. Eventually, the responsibility for understanding and maintaining inconsistent identifiers is delegated to software maintainers. This implies that it can be difficult to correct inconsistencies once they have happened since software maintainers have most of the responsibility to change identifiers and these often cannot be corrected if the maintainers miss the inconsistent identifiers. The *Etc.* section in the figure includes ‘sometimes modify the inconsistent identifiers regardless ownership’.
- *Why do you think such inconsistency happens?:* All participants agreed that inconsistency occurs as a result of human factors. Expressions for concepts that can vary depending on time, background knowledge, and so on. These can be different, even for a single person, depending on the time, meaning that inconsistency is inevitable. Although they maintain a glossary in the project, it is not observed as well. Some of the organizations build term management systems, also known as meta-data management system. However, all words cannot be maintained in the system and developers also feel inconvenient because they should register a term whenever they want to write a new term in the source code.

Questions on the Usefulness of our Approach



(a) Perception of Inconsistency

(b) Treatment of Inconsistency

(c) Usefulness of Our Approach

(d) Support for Software Lifecycle

Fig. 12: Analysis of the semi-structured interview.

- *How useful is our approach for detecting inconsistent identifiers?:* The participants stated that our approach is useful, as shown in Figure 12(c) and the results are acceptable. In addition, regardless of post-action after detecting inconsistent identifiers, they said that inspecting the inconsistency of identifiers was valuable in order to check the current quality of the entire source code. Although they did not think that inconsistent identifiers are not a severe issue when seeing the list of them at the first time, statistical information and reasons for each inconsistency from our approach made them realize it as a severe issue. Also, their feedback says that our tool *CodeAmigo* integrated with Eclipse provides good accessibility for software developers who spend their time writing source code.
- *When is our approach useful during the software life cycle?:* They said that it is particularly useful during inspection, peer review of the source code, and code auditing during implementation (see Figure 12(d)). Since our approach provides statistical usage results for all words in the source code, it enables the reviewer to judge the right choice of words. Although the subjects did not want to correct programs that belong to other developers, they stated that they will try to correct inconsistent identifiers detected using our approach during the review and auditing phases. Consequently, we could understand that our tool is more applicable to the review and code auditing phase in the software life cycle.
- *Have you ever used the tool checking inconsistent identifiers?:* Most of the subjects had not used such a tool that could check for inconsistent identifiers. Some of them had used CheckStyle [41] to check whether their source code adhered to a set of coding standards, or FindBug [42] to inspect the source code for potential defects. In addition, in-house term management systems that they built could manage terms used in the source code

while only managing domain-specific words in their application domains without managing various terms of the general computer domain. They feel that registering new words was too cumbersome when writing source code. Thus, they stated that our approach is suitable for the purpose of detecting inconsistent identifiers that they had never used before.

After interviewing the practitioners with the prepared questions, we freely discussed recurring issues related to inconsistency in the source code and to possible solutions for such issues. The following give a summary of our free discussion:

- *Relationships between Understandability and Inconsistency:* Understandability indicates how well readers can understand the source code and the intention of an identifier written by the original authors. Inconsistency is one of the causes that prevents understandability. Thus, it is important to remove inconsistent identifiers even though it is not the only essential activity that can improve understandability. They insisted that it must be integrated with checking for the code conventions.
- *Other types of Inconsistencies:* They did not suggest any new type of inconsistency for the identifiers. Instead, they proposed inconsistency in the code section management. For example, a developer writes a class with attributes in the upper part of a class while another developer places the attributes at the bottom of the class. Improving this in line with improving understandability of the source code. In addition, they suggested there could be critical inconsistency between the design model or the design documents and the source code. These might be important research issues.

4.4 Threats to Validity

Construct Validity: The identifier inconsistency may not have a strong correlation to software maintenance since there are many other aspects that influence software maintainability. To justify this issue, we collected and summarized several real examples from several issue tracking systems for open-source projects throughout Sections 1 and 2. These examples can show that identifier inconsistency may affect software maintenance

Contextual information can alleviate inconsistency issues. For example, differentiating the full signature of the methods "`retrieveUserId(String database)` and `searchUserID(String query)`" can be easier than for a simplified signature "`retrieveUserId()` and `searchUserID()`". However, the contextual information may not be helpful in some cases, as shown in Issue HBASE-584[23]: `filter(Text rowKey)`, `filter(Text rowKey, Text colKey, byte[] data)`, and `filterNotNull (SortedMap<Text, byte[]> columns)`. Evaluating the impact of the contextual information with inconsistency detection remains as a future task.

Content Validity: Four types of identifier inconsistencies defined in Section 2.2 might not completely identify all areas of inconsistency. There can be several different type of inconsistencies. For example, we can define "behavior inconsistency" as that which indicates

[23] https://issues.apache.org/jira/browse/HBASE-584

that the method or class names may be inconsistent with the behavior of the corresponding method or class.

Internal Validity: Different developers might identify inconsistent identifiers differently due to their different understanding of programs. To manage such cases, we analyzed the detection results from several developers through intersection and union sets as shown in Section 4.3.2. In addition, different threshold values for our approach might show different detection results. To alleviate this problem, we have conducted a sensitivity analysis to define the threshold values through a preliminary study using Apache Ant, as shown in Section 4.1.

When the participants verified the detection results on our web-site, we asked them to uncheck a box if our detection was not valid. We supposed that there were more valid detections than invalid ones, and such a design is intended to facilitate faster validation due to the massive amount of manual checking (3,826 detections). The participants agreed with our experimental method for the same reason. Although this method ("checked" by default) might be biased, the opposite method ("unchecked" by default) can be biased as well.

External Validity: Our approach may show different results depending on our subjects or for closed-source projects. Such programs may have significantly different naming conventions. For example, those programs can frequently use domain-specific words that can be detected as inconsistencies through our approach. In addition, different developers may have different criteria to identify inconsistencies. Thus, the result of our study shown in Section 4 can be different if different subjects participated in the study.

5 Related Work

This section presents prior work that targeted the detection of inconsistent identifiers in source code from two different perspectives. We first introduce existing concepts that handle inconsistent identifiers with respect to Semantic, Syntactic and POS inconsistencies. Then, technical comparisons between the previous approaches and ours are presented in turn. In the last section, we compare detections from the previous tool and CodeAmigo.

5.1 Inconsistency of Source Code Identifiers

Several previous studies handle inconsistencies of source code identifiers. Deiβenböck and Pizka [1] formally defined the inconsistency of source code identifiers. They divided inconsistencies into *homonyms* where an identifier is mapped to two concepts, and *synonyms* where more than two identifiers are mapped to a single concept. For example, the `File` identifier can be mapped into the two concepts `FileName` and `FileHandle` so that it can defined as a homonym. On the other hand, `account number` and `number` indicate an account number together, which is a synonym. Their synonym is conceptually the same with our semantic inconsistency. However, we added POS-constraints into the inconsistency to improve the preciseness when searching for appropriate synonyms, which was motivated by the use of general natural language dictionaries, such as Oxford [24] and Collins Cobuild [25].

In addition, the word-POS inconsistency is in line with the concept of a *homonym* [1]. This is because different POSes for a word indicate different concepts (meanings). These are limited to a word discovered in the natural language dictionary. Such a definition is inspired by Caprile and Tonella's observation [21], developers tend to consistently use a single POS of a word throughout a project even though the word has diverse POSes.

Deißenbock and Pizka [1], and Lawrie *et al.* [2] defined the term inconsistency as a mapping of a single term to more than two concepts. However, the different POSes of a word indicate different concepts (meaning or senses). For example, the word *sleep* as a noun generally indicates the state while it often represents a execution behavior as a verb. Thus, we defined a word-POS inconsistency. Note that word-POS inconsistency is limited to words in the natural language dictionary.

Abebe *et al.* introduced an *Odd Grammatical Structure* as one of the lexicon bad smells [5]. They checked if a specific POS existed in a class, method or attribute identifiers to detect a lexicon bad smell. For example, `Compute` as a class identifier is a lexicon bad smell because the class identifier contains a verb without any nouns. However, their missing parts in defining the grammatical rules of the identifiers when naming identifiers is that 'class names are generally a noun as well as a noun phrase'(see [43]). This indicates that checking if the entire identifier observes the phrase-POS rule is also valuable for developers because many identifiers are composed of more than two words. A phrase-POS inconsistency of our approach is similar to Abebe *et al.*'s odd grammatical structure [5], but it focuses more on the grammatical constraints of the entire identifier.

Hughes stressed the importance of spell checking in source code [6]. Eclipse, as the one of the most popular editing tools, also embeds a *Spell Checker* feature as a default, and it contains a language dictionary for support. The dictionary can be changed into others, such as SCOWL[26], as a custom word dictionary for the spell checker. Such spell checkers however do not work when two misspelled words are discovered in the dictionary (e.g., `states` and `status`). Since the syntactic inconsistency presented in the paper handles similar character sequences of two words, it can detect structural-inconsistent identifiers, including misspelled-words regardless of the existence in the dictionary. It also detects words that might be confused due to their similar letter sequences.

5.2 Detecting Inconsistent Identifiers

In order to handle inconsistent identifiers, diverse guidelines have been introduced by industry and academia. In industry, Sun Microsystems (acquired by Oracle) suggested naming conventions to guide identifier naming by using grammatical rules [14]. Also, various

[24] Oxford Dictionary, http://www.oxforddictionaries.com/

[25] Collins Cobuild Dictionary:http://www.collinsdictionary.com/dictionary/english

[26] SCOWL: http://wordlist.aspell.net/

industrial practitioners have placed an emphasis on careful naming of the source code elements [3, 17, 44]. Most of them stated that identifiers in a program should be used in a consistent manner.

In academia, Lawrie *et al.* [15] tried to analyze the trends for code identifiers by establishing a model for measuring the quality of the identifiers, and then mining 186 programs over three decades. Their statistical findings indicate that 1) modern programs contain higher quality identifiers, 2) the quality of open and proprietary source identifiers is different, and 3) a programming language does not largely affect the quality of the identifiers. Additionally, they mentioned that Java uses relatively high quality identifiers, compared to other programming languages, thanks to an early encouragement of coding and naming conventions.

Deiβenböck and Pizka [1] formally defined the inconsistencies of source code identifiers, as mentioned above. While it is valuable to introduce this issue at first, it has shortcomings in that developers should manually map the identifiers to the concepts. In order to handle the shortcoming of Deiβenböck and Pizka's approach, Lawrie *et al.* [2] used the patterns of a letter sequence of identifiers and WordNet [18]. They figured out that an identifier exists in the other identifiers, which is a way of detecting a homonym. For example, the identifier `FileName` and `FileHandle` have `File` as a part of the identifiers. In addition, they automatically searched synonyms in WordNet. Although they applied WordNet to find semantically similar words, they did not analyze the POS of each word that composed an identifier. This makes the scope of the synonyms very diverse, which must decrease the precision when detecting synonyms.

Abebe *et al.* [5] proposed *Lexicon Bad Smell* indicating inappropriate identifiers in terms of the lexicon, and they presented a tool support *LBSDetector* in order to automatically detect improper identifiers. Among the lexicon bad smells, the *Odd grammatical structure* smell is similar to the Phrase-POS inconsistency, which indicates issues, such as when class identifiers do not contain a noun, attributes that contain verbs, and methods that do not start with a verb. However, such a method can cause false alarms. For example, `length()` and `size()`, as method identifiers, and `debug` and `warn`, as attribute identifiers, are detected as odd grammatical structure bad smell. In our approach, we have followed the Java naming convention without defining new grammatical rules, as introduced in Section 2. Also, we built a Code Dictionary that stores idiom identifiers, such as `length()` and `size()`, which commonly violate Java naming conventions but are acceptable.

Abebe and Tonella [9] and Falleri *et al.* [10] built an ontology containing concepts and their relationships by using source code identifiers. They first separated the words from the identifiers, composed a sentence according to their rules, parsed the sentence with a natural language parser, and defined them as knowledge. Using such knowledge, the developer names the identifier. This approach is similar to our approach in terms of using an NLP parser, but they are not intended to detect inconsistent identifiers.

Abebe and Tonella [11] then introduced an automated approach that exploits ontological concepts and relations extracted from the source code in order to suggest identifiers on the fly. The purpose of their approach is to help developers name identifiers by automatically suggesting related concepts such as auto-completion. While this approach may be helpful when developing the system, it can be hardly used by software maintainers or reviewers for verification because it is not intended to support scanning all identifiers and detecting an inconsistent use of the terms and their relations throughout the project. Their tool has not been developed, yet it is described in the paper, which is not available to use.

Some researchers have tried to extract vocabulary from the source code [7, 8]. Lawrie *et al.* [8] suggested an approach to normalizing vocabulary from source code in order to mitigate an expression gap between software artifacts written in a natural language and source code written in a programming language. They separated an identifier into several possible soft-words and computed the similarity between the soft-words and the natural language words in external dictionaries based on the wild-card expansion algorithm [15]. This is valuable to map diverse acronyms and abbreviations in the source code into concepts that natural language can apply for Information Retrieval techniques [31] when mining source code.

Delorey *et al.* [7] also proposed an approach that builds a corpus from source code by defining four levels to denote identical words. They classified the words according to the same letter sequence as POS, and analyzed the frequency of the occurrence of the word in the JDK 1.5 source code. Their concept for 'word' and POS are mapped into an identifier and a type of an identifier (e.g., class, method or field) respectively. It has a limitation in handling a word as a constituent of an identifier.

For method identifiers, Host and Ostvold [12] investigated verb usage in the source code and built a pattern for a verb starting with the verb (e.g., `contains-*`). The pattern can include 'returns int/string', 'create string/custom objects', 'throws exceptions', etc. They extracted it from the source code and defined it as a pattern of a specific verb of a method, and then they indicate a violation of the pattern as a *naming bug*. While it may contribute to consistent use of a verb of a method throughout Java based applications, it is not applicable for detection of inconsistent identifiers in a single project. Also, it only focuses on the method identifiers without considering other source code elements.

Arnaoudova *et al.* [45] defined a new term *Linguistic Anti-Patterns* to identify recurring poor practices in naming and choosing identifiers. They categorized the anti-pattern into six sub-groups: three for the methods and the other three for the attributes. For example, the methods that started with the verb *get* (e.g., `getImageData`) and not just returned an attribute value were classified into the 'do more than it says' group, which is one of the categories for the linguistic anti-patterns. This is in the line with Host and Ostvold' research, and it is valuable for the detection of the consistent use of the terms throughout the project.

Table 10 summarizes prior studies in terms of the software life-cycle support, the types of inconsistency and tool supports after carefully selecting research that contributes to detecting or alleviating inconsistent identifiers. Since POS are not considered for searching synonyms, researches for SEM has been evaluated with '+'. Deiβenböck and Pizka [1] and Lawrie *et al.* [2] evaluated with '+' for POS-WORD, as homonyms in their research were related to Word-POS Inconsistency. Abebe and Tonella [11] presented an approach that can be used when writing code which supports automatic suggestion of appropriate words, thus we evaluated it as '+' for SEM and SYN. However, the tool has not been fully developed yet. Spell Checkers can contribute to an alleviation of SYN and can be used during code review and code writing. Our approach covers relatively diverse inconsistencies as compared to others.

Table 10: Comparison with previous work (++: well-supported, +: supported, ○: not supported).

Research	Life-cycle Support	SEM	SYN	POS-WORD	POS-PHR	Tool Support
Deiβenböck and Pizka [1]	Code Review	+	○	+	○	++
Lawrie *et al.* [2]	Code Review	+	○	+	○	++
Abebe *et al.* [5]	Code Review	○	○	○	+	++
Abebe and Tonella [11]	Code Writing	+	+	○	○	○
Spell Checkers [6]	Code Review/Writing	○	+	○	○	++
Our Approach	Code Review	++	++	++	++	++

5.3 Comparing the Previous Tool with Our Approach

This section presents comparison between our approach and an existing technique. Among the previously introduced techniques, we selected *LBSDetector* developed by Abebe *et al.* [5] because it is publicly available[27] and provides a similar feature with phrase-POS inconsistencies of this paper: *Odd Grammatical Structure* bad smells. Rules for detecting the bad smells include:

- Class identifiers should contain at least one noun and should not contain verbs,
- Method identifiers should start with a verb, and
- Attribute identifiers should not contain verbs.

By using LBSDetector, we extracted bad smells for odd grammatical structure in *JUnit*. Then, we compared the results with manual detection results by human subjects (see Section 4.3.2) and those of our approach shown in Section 4. In particular, only phrase-POS inconsistencies detected by our approach were used since the inconsistencies are conceptually compatible with odd grammatical structure defined in LBSDetector [5].

Initially, *LBSDetector* found 426 identifiers as bad smells for odd grammatical structure while our approach detected 148 identifiers for phrase-POS inconsistency. We removed redundant identifiers to compare them with inconsistencies detected by human subjects. As a result, 303 and 123 identifiers were obtained for each approach. Then, we conducted pairwise comparison between the results by human subjects and each approach with respect to precision, recall, and F-measure as shown in Table 11.

Table 11: Precision, Recall & F-Measure of our approach and LBSDetector

Tools	Precision	Recall	F-Measure
CodeAmigo	0.73 (90/123)	0.67 (90/133)	0.70
LBSDetector	0.34 (106/303)	0.79 (106/133)	0.48

[27] `Lexicon BadSmell Wiki`: http://selab.fbk.edu/LexiconBadSmellWiki

The precision of our approach was higher than *LBSDetector*. Note that our approach figures out the POS of an identifier as a whole by parsing it while LBSDetector only finds out whether a specific POS is used in an identifier. The different ways of POS interpretation might lead to the different precision. The recall of *LBSDetector* was higher than that of our approach. This might be caused by less strict detection rules of LBSDetector than our approach.

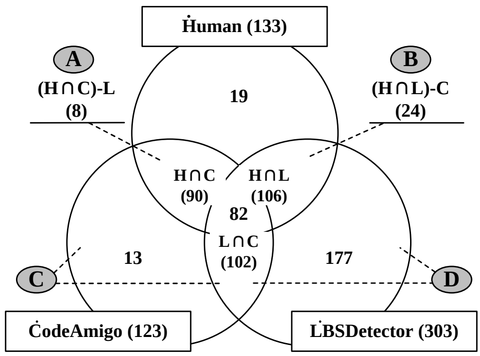


Fig. 13: Detection results by our approach, *LBSDetector*, and human subjects for JUnit. "Phrase-POS inconsistency" of our approach and "odd Grammar structure bad smells" of LBSD detectors were considered since these are only compatible inconsistency concepts between two approaches.

We examined the detection results to figure out the intersection and disjoint sets between three different detection sources. The result of examination is shown in Figure 13. In the figure, we focused on the disjoint sets (i.e., the area *A* and *B*) of CodeAmigo and LBSDetector detection results in the Human set. In the *A* area, four method identifiers, *forValue*, *optionallyValidateStatic*, *or* and *allOf*, were detected by human subjects and CodeAmigo together as phrase-POS inconsistency while LBSDetector could not detect them. In addition, four class identifiers, *Assume*, *Ignore*, *Assert*, *Each*, were not detected by LBSDetector. This might be caused by missing information of WordNet and wrong NLP parsing by Minipar [26], which is used in LBSDetector.

For the area *B*, CodeAmigo could not detect three identifiers *Protectable*, *comparator*, and *errors* due to WordNet's missing information. Ten identifiers were not detected by our approach (e.g., *RunWith* and *defaultComputer*) due to the Stanford parser's errors. In addition, CodeAmigo could not detect 11 custom naming conventions (e.g., attributes starting with *f*- that JUnit denotes it as a *field*) but LBSDetector could. For example, the attribute identifiers *fUnassigned*, *fExpected*, *fActual* were only detected by LBSDetector.

There were 33 false positives of our approach (see the *C* area in Figure 13) such as *testStarted()*, *testFailed()*, *fireTestRunStarted()*, and *formatClassAndValue()*, each of which

Table 12: Comparison of *CodeAmigo* with *LBSDetector* (*C*-CodeAmigo, *L*-LBSDetector, *I*-Intersection, *Lu*-Lucene, *JM*-JMeter, *JHD*-JHotDraw, *SH3D*-Sweet Home 3D, *Avg*-Average)

	Lu	*Ant*	*JM*	*JHD*	*SH3D*	*C*	*Avg*
C	307	363	239	223	138	26	216
L	1,711	2,511	3,633	2,116	1,970	404	2,057.5
I	157	277	82	112	58	11	116.1
I/C	51.1%	76.3%	34.3%	50.2%	42.0%	42.3%	49.4%
I/L	9.2%	11.0%	2.3%	2.3%	2.9%	2.7%	5.6%

are caused by wrong NLP Parsing. LBSDetector detected 197 false positives (the *D* area). For example, class identifiers such as *Rule*, *TestClass*, *TestSuite*, and method identifiers including *getIncludedCategory()*, *hasItem()*, *compact()*.

These false positives of both approaches were mostly caused by wrong NLP parsing. This indicates that template-based preprocessing of an identifier is necessary before applying an NLP parser to identifiers [9]. In addition, some of false positives such as *setUp()* and *main()* were found in idioms of Code Dictionary. The gap of the false positives can be understood as the phrase-POS inconsistency was more helpful than odd grammatical structure to reduce false positive but less rigid rules can detect violations of custom naming conventions.

We also compared inconsistency results of odd grammatical structure bad smells detected by LBSDetector and phrase-POS inconsistencies detected by our approach for six remaining projects and summarized the results in Table 12. All redundant identifiers had been removed. 49.4% of inconsistency results by CodeAmigo were also discovered in those of LBSDetector on average while 5.6% of inconsistencies by LBSDetector were found in those of CodeAmigo. This implies that LBSDetector resulted in a higher number of false positives.

6 Conclusion

In this paper, we have presented an approach, based on a Code Dictionary, which detects inconsistent identifiers. This approach first builds a code dictionary containing domain words with dominant POS information and idioms that are frequently discovered in the popular open source projects. The approach then detects semantic, syntactic, and POS inconsistent identifiers. The code dictionary helps filter out false alarms, and the evaluation results indicated that our approach accurately detected inconsistent identifiers. 8 out of 10 identifiers detected by our approach were found to be correctly identified, according to the human subjects. In addition, an interview with six developers confirmed that our approach was helpful for automatically finding inconsistent identifiers. By using this approach, developers can identify inconsistent identifiers, and can therefore improve software maintainability. For future work, we are planning to survey diverse types of inconsistencies in the source code in order to improve software maintenance.

Acknowledgements.
This paper was supported by research funds of Chonbuk National University in 2014.

References

1. F. Deiβenböck and M. Pizka. Concise and Consistent Naming. In *Proceedings of International Workshop on Program Comprehension(IWPC)*, pages 261–282, St. Louis, MO, USA, 2005.
2. D. Lawrie, H. Field, and D. Binkley. Syntactic Identifier Conciseness and Consistency. In *Proceedings of IEEE International Workshop on Source Code Analysis and Manipulation(SCAM)*, pages 139–148, Philadelphia, Pennsylvania, USA, 2006.
3. R.C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall; 1 edition, 2008.
4. Y. Higo and S. Kusumoto. How Often Do Unintended Inconsistencies Happen?-Deriving Modification Pattern and Detecting Overlooked Code Fragments-. In *Proceedings of the 28th International Conference on Software Maintenance*, pages 222 – 231, Trento, Italy, 2012.
5. S.F. Abebe, S. Haiduc, P. Tonella, and A. Marcus. Lexicon Bad Smells in Software. In *Proceedings of Working Conference on Reverse Engineering*, pages 95–99, Antwerp Belgium, 2008.
6. E. Hughes. Checking Spelling in Source Code. *IEEE Software, ACM SIGPLAN Notices*, 39(12):32–38, 2004.
7. D.P. Delorey, C.D. Kutson, and M. Davies. Mining Programming Language Vocabularies from Source Code. In *Proceedings of the 21st Conference of the Psychology of Programming Group(PPIG)*, London, United Kingdom, 2009.
8. D. Lawire, D. Binkley, and C. Morrel. Normalizaing Source Code Vocabulary. In *Proceedings of the 17th Working Conference on Reverse Engineering*, pages 3–12, Boston, USA, 2010.
9. S.L. Abebe and P. Tonella. Natural Language Parsing of Program Element Names for Concept Extraction. In *Proceedings of International Conference on Program Comprehension(ICPC)*, pages 156–159, Minho, Portugal, 2010.
10. J. Falleri, M. Lafourcade, C. Nebut, V. Prince, and M. Dao. Automatic Extraction of a WordNet-like Identifier Network from Software. In *Proceedings of International Conference on Program Comprehension(ICPC)*, pages 4–13, Minho, Portugal, 2010.
11. S. Abebe and P. Tonella. Automated identifier completion and replacement. In *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR)*, pages 263 – 272, Genova, Italy, 2013.
12. E. W. Host and B.M. Ostvold. Debugging Method Names. *Proceedings of the 23rd European Conference on Object-Oriented Programming, Lecture Notes in Computer Science*, 5653(1):294–317, 2009.
13. S. Lee, S. Kim, J. Kim, and S. Park. Detecting Inconsistent Names of Source Code Using NLP. *Computer Applications for Database, Education, and Ubiquitous Computing Communications in Computer and Information Science*, 352(1):111–115, 2012.
14. Code Conventions for the Java Programming Language: Why Have Code Conventions, Sun Microsystems,1999. http://www.oracle.com/technetwork/java/index-135089.html.
15. D. Lawrie, H. Feild, and D. Binkley. Quantifying identifier quality: an analysis of trends. *Empirical Software Engineering*, 12(4):359–388, 2007.
16. N. Madani, L. Guerroju, M.D. Penta, Y. Gueheneuc, and G. Antoniol. Recognizing Words from Source Code Identifiers using Speech Recognition Techniques. In *Proceedings of 14th European Conference on Software Maintenance and Reengineering(CSMR)*, pages 68–77, Madrid, Spain, 2010.
17. P. Goodliffe. *Code Craft: The Practice of Writing Excellent Code*. No Starch Press, 2006.
18. WordNet: A lexical database for English, Home page,2014. http://wordnet.princeton.edu/.
19. R.N. Haber and R.M. Schindler. Errors in proofreading: Evidence of Syntactic Control of Letter Processing? *Journal of Experimental Psychology: Human Perception and Performance*, 7(1):573–579, 1981.
20. A.F. Monk and C. Hulme. Errors in proofreading: Evidence for the Use of Word Shape in Word Recognition. *Memory & Cognition*, 11(1):16–23, 1983.
21. B. Caprile and P. Tonella. Nomen Est Omen: Analyzing the Language of Funtion Identifiers. In *Proceedings of Working Conference on Reverse Engineering*, pages 112–122, Altanta, Georgia, 1999.

22. The Stanford Parser: A statistical parser, Home page,2014. http://nlp.stanford.edu/software/lex-parser.shtml.
23. Apache OpenNLP Homepage,2014. http://opennlp.apache.org/.
24. D. Binkley, M. Hearn, and D. Lawrie. Improving Identifier Informativeness using Part of Speech Information. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, pages 203 – 2006, New York, NY, 2011.
25. S. Guapa, S. Malik, L. Pollock, and K. Vijay-Shanker. Part-of-Speech Tagging of Program Identifiers for Improved Text-Based Software Engineering Tools. In *Proceedings of 21st International Conference on Program Comprehension (ICPC)*, pages 3 – 12, San Francisco, CA, 2013.
26. MINIPAR Homepage,2014. http://webdocs.cs.ualberta.ca/ lindek/minipar.htm.
27. K Toutanova, D. Klein, C. Manning, and Yoram Singer. Feature-Rich Part-of-Speech Tagging with a Cyclic Dependency Network. In *Proceedings of HLT-NAACL*, pages 252 – 259, 2003.
28. The Penn Treebank Project, 2013. http://www.cis.upenn.edu/ treebank/.
29. A. Budanitsky and G. Hirst. Evaluating WordNet-based Measures of Lexical Semantic Relatedness. *Computational Linguistics*, 32(1):13–47, 2006.
30. V. I. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady*, 10(8):707–710, 1966.
31. W.B. Frakes and R. Baeza-Yates. *Information Retrival : Data Structures and Algorithms*. Englewood Cliffs, J.J.: Prentice-Hall, 1992.
32. Apache Lucene Homegage,2013. http://lucene.apache.org/core/.
33. Apache Ant Homepage,2013. http://ant.apache.org/.
34. Apache JMeter Homepage,2013. http://jmeter.apache.org/.
35. JUnit Homepage,2013. http://www.junit.org/.
36. JHotDraw 7 Homepage,2013. http://www.randelshofer.ch/oop/jhotdraw/.
37. Sweet Home 3D Homepage,2013. http://sourceforge.net/projects/sweethome3d.
38. D. Klein and C.D. Manning. Accurate Unlexicalized Parsing. In *Proceedings of the Meeting of the Association for Computational Linguistics*, pages 423–430, Sapporo, Japan, 2003.
39. Code Amigo Validation WebPage,2014. http://54.250.194.210/.
40. D.M. Powers. Evaluation: From Precision, Recall and F-Factor to ROC, Informedness, Markedness & Correlation. *Journal of Machine Learning Technologies*, 1(1):37–63, 2011.
41. Eclipse-CS Check Style Homepage,2013. http://eclipse-cs.sourceforge.net/.
42. Find Bugs in Java Programs Homepage,2013. http://findbugs.sourceforge.net/.
43. Joshua Bloch. *Effective Java Programming Language Guide*. Sun Microsystems, 2001.
44. J. Bolch. *Effective Java (2nd Edition)*. Addison-Wesley; 2 edition, 2008.
45. V. Arnaoudova, M.D. Penta, G. Antoniol, and Y. Gueheneuc. A New Family of Software Anti-Patterns: Linguistic Anti-Patterns. In *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR)*, pages 187 – 196, Genova, Italy, 2013.

Appendix A. List of Domain Word POSes and Idioms

Table 13: Domain words with the dominant POS information extracted from the API document of projects with the parameter $T_{WO} = 100$ and $T_{PR} = 0.8$ (~~Strikeout~~ indicates a word evaluated as invalid in the preliminary study. The precision is computed as $176/191 = 0.921$).

Word	POS	Frequency	Word	POS	Frequency	Word	POS	Frequency	Word	POS	Frequency
abstract	noun	189	entry	noun	147	manager	noun	146	selection	noun	207
~~accessible~~	noun	226	error	noun	299	map	noun	189	separator	noun	110
action	noun	322	event	noun	299	max	noun	197	server	noun	120
add	verb	712	exception	noun	557	menu	noun	248	service	noun	138
annotation	noun	318	factory	noun	359	message	noun	204	set	verb	4946
annotations	noun	185	field	noun	244	method	noun	131	sheet	noun	147
array	noun	206	file	noun	689	mode	noun	173	size	noun	401
attribute	noun	243	fill	noun	91	model	noun	250	slide	noun	96
auto	noun	138	filter	noun	223	mouse	noun	123	source	noun	156
background	noun	124	~~first~~	noun	116	names	noun	121	spacing	noun	119
bar	noun	172	flag	noun	138	~~no~~	noun	99	~~split~~	noun	89
base	noun	110	flags	noun	106	node	noun	195	~~sprm~~	noun	124
~~basic~~	noun	227	focus	noun	143	num	noun	146	~~st~~	noun	222
bean	noun	230	font	noun	212	number	noun	228	state	noun	140
block	noun	123	format	noun	299	object	noun	302	stream	noun	328
border	noun	445	frame	noun	143	offset	noun	142	string	noun	397
bottom	noun	122	get	verb	8146	output	noun	206	style	noun	181
box	noun	162	~~gr~~	noun	119	page	noun	151	~~supported~~	noun	95
button	noun	224	~~grammatical~~	noun	134	paint	verb	190	system	noun	148
byte	noun	101	group	noun	130	pane	noun	293	tab	noun	145
cell	noun	461	gui	noun	124	panel	noun	101	table	noun	303
change	noun	138	handler	noun	291	parameter	noun	111	tag	noun	284
char	noun	138	header	noun	235	parser	noun	112	target	noun	105
character	noun	105	helper	noun	179	part	noun	108	task	noun	117
chart	noun	138	~~hssf~~	noun	113	path	noun	295	test	noun	168
child	noun	144	html	noun	118	pattern	noun	105	text	noun	774
chooser	noun	124	http	noun	124	policy	noun	159	thread	noun	147
class	noun	383	icon	noun	236	position	noun	113	time	noun	229
code	noun	122	id	noun	424	properties	noun	181	title	noun	113
color	noun	408	image	noun	218	property	noun	531	tool	noun	102
column	noun	228	impl	noun	96	quaqua	noun	238	top	noun	135
command	noun	125	index	noun	308	reader	noun	145	tree	noun	366
component	noun	236	info	noun	251	record	noun	398	type	noun	942
content	noun	182	input	noun	253	ref	noun	117	ui	noun	464
context	noun	259	int	noun	135	reference	noun	98	~~unknown~~	noun	97
control	noun	106	internal	noun	124	relations	noun	118	url	noun	122
core	noun	219	is	verb	1699	remove	verb	333	use	noun	129
~~count~~	noun	261	item	noun	114	report	noun	111	user	noun	120
create	verb	809	java	noun	141	request	noun	100	value	noun	525
data	noun	491	key	noun	428	resource	noun	183	version	noun	137
date	noun	160	label	noun	144	result	noun	141	~~vertical~~	noun	102
default	noun	599	layout	noun	154	right	noun	192	view	noun	151
display	noun	92	length	noun	120	root	noun	112	vk	noun	188
document	noun	224	line	noun	252	row	noun	323	width	noun	166
editor	noun	134	list	noun	302	sample	noun	94	window	noun	120
element	noun	265	listener	noun	504	sampler	noun	107	word	noun	160
end	noun	167	location	noun	117	scroll	noun	137	xml	noun	249
engine	noun	125	log	noun	92	~~selected~~	noun	102			

Table 14: Idiom identifiers extracted from the API document of projects listed in Table 1, where $T(FO_{fmw}) = 2$, $T(FO_{cls}) = 2$, $T(FO_{att}) = 2$, and $T(FO_{met}) = 10$.

Identifier	Type	Identifier	Type	Identifier	Type
abs	Met	intValue	Met	pow	Met
actionPerformed	Met	itemStateChanged	Met	preferredLayoutSize	Met
ALL	Attr	keyPressed	Met	previous	Met
ANY	Attr	keyReleased	Met	propertyChange	Met
available	Met	keySet	Met	random	Met
BOOLEAN	Attr	lastIndexOf	Met	readDouble	Met
copyOfRange	Met	layoutContainer	Met	reflectionHashCode	Met
debug	Attr	length	Met	~~requestFocus~~	Met
DEBUG	Attr	LONG	Attr	setup	Met
decrement	Met	longValue	Met	shortValue	Met
DEFLATED	Attr	~~lookup~~	Met	stateChanged	Met
DELETE	Attr	main	Met	substring	Met
element	Met	markSupported	Met	text	Met
elements	Met	max	Met	treeNodesChanged	Met
entrySet	Met	maximumLayoutSize	Met	treeNodesInserted	Met
error	Met	min	Met	treeNodesRemoved	Met
FALSE	Attr	minimumLayoutSize	Met	treeStructureChanged	Met
fatal	Met	mouseClicked	Met	validIndex	Met
first	Met	mouseDragged	Met	valueChanged	Met
~~getLayoutAlignmentX~~	Met	mouseEntered	Met	valueOf	Met
~~getLayoutAlignmentY~~	Met	mouseExited	Met	values	Met
~~getX~~	Met	mouseMoved	Met	verbose	Attr
~~getY~~	Met	mousePressed	Met	VERBOSE	Attr
IGNORE	Attr	mouseReleased	Met	WARN	Attr
increment	Met	newInstance	Met	warning	Met
indexOf	Met	next	Met	writeDouble	Met
info	Met	notEmpty	Met		