Lexicon Bad Smells in Software

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

[1] FBK-irst
38123 Povo,
Trento, Italy

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

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

Abstract

We introduce the notion of "lexicon bad smell", which parallels that of "code smell" and indicates some potential lexicon construction problems that can be addressed through refactoring (e.g., renaming). We created a catalog of lexicon bad smells and we developed a publicly available suite of detectors to locate them. The paper presents a case study in which we used the detectors on two open-source systems. The study revealed the main challenges faced in detecting the lexicon bad smells.

1 Introduction

Source code is not written only to be run by a computer; other developers will also read it, and will need to understand and modify it. One of the approaches commonly followed to achieve a quality internal documentation (i.e., comments and identifiers) is to use standard or adapt naming conventions. However, enforcing these conventions and checking if they are strictly followed is difficult.

Locating problems with the internal documentation, as early as they are introduced in the source code, has two main advantages: maintaining the quality of the source code and preventing others from misunderstanding the code functionality. In this regard, we introduce the notion of "lexicon bad smell".

A "lexicon bad smell" is a concept similar to that of a "code smell" and it refers to potential lexicon construction problems, which could be solved by means of refactoring (typically renaming) actions. It is a relative notion, whose definition and use depends on the idiosyncrasies of the project, programming environment, skills of programmers, etc. Hence, a bad smell in the source code of a system might not be considered a bad smell in another.

We have defined a catalog of lexicon bad smells (not all new) and created a Wiki[1] that maintains it. Currently, the Wiki contains a preliminary list of bad smells, but we plan to extend this collection and open the access to the Wiki to the research community. We have also implemented a suite of tools for detecting most of the smells, which is publicly available through the Wiki. In this paper, we focus on five non-trivial smells and make a preliminary evaluation of the precision of their detectors, by conducting a case study on two open source systems. The evaluation provides important insights on directions for future improvements.

2 Terminology

Entities are used to refer to classes, and class member elements (i.e. class attributes, and methods). The name of an entity is called an ***identifier***. Identifiers are formed by one or more ***terms***, which are sequences of characters. ***Hard terms*** are separated from adjacent terms by the use of CamelCase, "_", or other non-literals. Those which are not clearly separated from adjacent terms are called ***Soft terms***. Terms can be one of the following: word, abbreviation of a word, contraction of a word, acronym, a sequence of characters that is none of the above (token). A ***word*** is the smallest free form in a language, usually consisting of a root or stem and zero or more affixes.

3 Lexicon bad smell catalog

We present here five from the lexicon bad smells listed in the Wiki. We studied these bad smells at the level of class, attribute, and method identifiers.

3.1 Odd grammatical structure

Description. The grammatical structure of an identifier is not appropriate for the specific type of entity it represents.

Symptoms. A syntactical rule concerning the construction of an identifier is not respected, such as:

- class identifiers should contain at least one noun and should not contain verbs;
- method identifiers should start with a verb;
- attribute identifiers should not contain verbs, etc.

[1] http://selab.fbk.eu/LexiconBadSmellWiki

1095-1350/09 $26.00 © 2009 IEEE
DOI 10.1109/WCRE.2009.26

Examples.

```
class Compute { //verb
  public void initialization(); //noun
}
```

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

Detector. For every class identifier, attribute identifier and method identifier in the system, the detector checks if the related structuring rules are followed. It uses the *Minipar*[2] English parser to determine the parts of speech for every identifier.

3.2 Term used to name both the whole and its parts

Description. The same term is used to represent a concept and its properties or operations.

Symptoms. A term is used to name a class and it appears also in some method or attribute identifier in the same class. This might indicate either ambiguous use of the term or redundancy.

Example.

```
class Account {
   int account;   // Ambiguous use
   void computeAccount();
   // Account is redundant information
 }
```

Exception: A static attribute, used to realize the singleton design pattern has usually the same identifier as the class. Constructor methods have the same name as the class.

Refactoring. Rename different entities so as to differentiate their role and/or avoid redundant information.

Detector. The detector identifies the last noun (when possible) or takes the last term of the class identifier and checks if it is used in attribute and/or method identifiers. The stems of the words are considered instead of the full form, in order to account for inflections.

3.3 Inconsistent identifier use

Description. A concept is not represented by identifiers in a consistent and concise way.

Symptoms. Since usually no formal concept–to-identifier mapping is available, we resort to the following symptom: all soft terms of an identifier are contained in the same order in another identifier of the same entity type and both identifiers are located inside the same container entity (e.g., class).

Example.

```
class Documents {
    private String absolute_path;
    private String relative_path;
    // path has ambiguous meaning
    private String path;          }
```

[2] http://www.cs.ualberta.ca/~lindek/minipar.htm

Refactoring. Rename identifiers to make them more concise and consistent.

Detector. The detector checks if an entire identifier is contained in another identifier of the same entity type (attributes or methods), inside the same container entity (a class).

3.4 Useless type indication

Description. The type of a variable is explicitly indicated in its identifier. This represents redundant information in the context of modern programming environments, which provide easy access to type information for all variables.

Symptoms. An identifier contains more than one term, one of which is the identifier's type name.

Example.

```
class Rental {
    // type in attribute name
    short key_short;
}
```

Exception. A static attribute used to realize the singleton design pattern; sequences of characters imposed by a naming convention, which denote the type of the variable (e.g., in the Hungarian notation, *i* is used in the identifiers of integer values).

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

Detector. The detector checks if attribute identifiers contain their type name.

3.5 Identifier construction rules

Description. The naming of an identifier does not follow a standard naming convention adopted in the system.

Symptoms. Some existing naming convention (or the prevalent naming convention, if none is explicitly documented) for identifiers is not respected.

Example.

```
class StudentInformation {
   private String fName;
   // should start with an f
   private String Address;
}
```

Refactoring. Restructure the identifier by following the adopted naming convention.

Detector. The detector verifies if the identifiers are constructed according to the predefined naming rules for each specific entity type.

4 Tools

Manual inspection of the source code to identify lexicon bad smells is a tedious and difficult task. Hence, we have developed a suite of tools, called *LBSDetectors* that automatically locate and report lexicon bad smells. The tools use different plug-ins and software components like *Minipar*, a parser for

identifying parts of speech of the terms composing an identifier (considered as a phrase); *PaWs*[3], a wrapper to *Minipar;* and *src2srcml*[4], for transforming the source code files into XML.

We currently implemented detectors for eleven of the bad smells described in the Wiki. In this paper, we present the evaluation of five of these tools, corresponding to the bad smells described in Section 3. The tools use customizable configuration files and thresholds, which makes them adaptable to specific needs in a particular software system.

5 Case study

The **goal** of the case study reported in this section is to evaluate the accuracy of the detectors, in order to identify those which require further elaboration or the definition of brand new analyses and heuristics.

The **question** descending from the case study's goal is: *Which bad smells are hard to find using the current detectors and how can the detectors be improved to perform better in these cases?* We computed and report on one **metric**, *precision*, to address the first part of the question, while the answer to the second part comes from arguments based on observations of the reported false positives. *Precision* is defined as the ratio between reported smells that are correct and total number of reported smells. Similar retrieval tasks also use *recall* to measure the effectiveness of retrieval tools. In our case we do not have any data about the total number of bad smells in the software systems considered; hence recall can not be computed.

In order to evaluate the precision of the detectors, we applied the following **guideline**: *A reported bad smell is a false positive if developers are not expected to be willing to take any action (e.g., renaming) to improve the "smelly" identifier.* Of course, sometimes it is hard to make a decision that reasonably parallels that of real developers. In those cases, we either contact the original developers and get an authoritative decision from them or report an approximate precision. The approximate precision is presented as a range computed by considering the ambiguous case as both false and true positive.

The open-source software systems analyzed in our case study are *Alice* (aliceinfo.cern.ch), which has 7,321 files (1,507,046 lines of text), and *WinMerge* (www.winmerge.org) which has 1,147 files (431,034 lines of text). Both are written in C++.

For some of the bad smells, the number of reported detections was very large. To make the manual assessment of precision feasible, in these cases we randomly selected a sample from the reported bad smells and made our assessments on the sample.

5.1 Odd grammatical structure

The detector for this bad smell has identified 26,973 bad smells (1,439 in class, 9,652 in attribute, and 15,884 in method identifiers) in Alice and 3,120 (242 in class, 686 in attribute, and 2,192 in method identifiers) in WinMerge. The precision obtained after the manual investigation of the results for 30 randomly selected entries (10 class, 10 attribute, and 10 method identifiers) in Alice was 6.67–10%, while for WinMerge it was 33–36.67%. Samples of the results with the evaluation are shown in Table 1.

Table 1. Sample results of the *odd grammatical structure* detector in Alice

Entity	Identifier	Evaluation FP=False Positive
class	FemtoShareQualityKTPairCut	Bad smell
class	HBTMonPyResolutionVsPtFctn	FP-nouns,specifiers
attribute	UseBoxDigits	FP-boolean value
method	Value	FP-implicit get

The false positives are mainly due to two reasons. First, the detector relies on the output of *Minipar* to identify the parts of speech of the terms in the identifier. In some cases (like in class identifier *HBTMonPyResolutionVsPtFctn*), *Minipar* is not able to identify the parts of speech of some of the terms. Second, the detector reported also the exceptional cases where verbs are not specified in method identifiers, but they are implicitly considered (e.g., *value* is implied to be *getValue*, as it returns the value of an attribute). Also, it is common that boolean attributes contain verbs (e.g., *UseBoxDigits*), in which case these should not be considered bad smells. The next generation of detectors will consider these cases.

5.2 Terms used to name both the whole and its parts

The detector for this bad smell has identified 2,957 lexicon bad smells in Alice and 768 in WinMerge. Of the total smells in Alice, 1,103 are found in attribute identifiers and 1,854 in method identifiers. In WinMerge, 120 are found in attribute identifiers and 648 in method identifiers.

To evaluate the precision of this detector, we have randomly selected 10 bad smells reported for attribute identifier-class identifier pairs and 10 for the method identifier-class identifier pairs. The precision of the tool for Alice is 75–90% and for WinMerge 75%. Most of the false positives reported in WinMerge are due to a common practice in testing to start identifiers of test methods with *test,* which in this case is also used in the class identifier where the method is

[3] http://ontoware.org/projects/paws/
[4] http://www.sdml.info/projects/srcml/

declared (see Table 2). The next release of the detector will consider these exceptions.

Table 2. Sample method identifiers that contain part of their class identifier in WinMerge

Method identifier	Class identifier	Evaluation
testRunTestFailure	TestCase***Test***	FP
OnUpdate***View***SwapPanes	MergeEdit***View***	Bad smell
testDetailsSome	Message***Test***	FP
Get***Version***Number	VSS***Version***	Bad smell

5.3 Inconsistent identifier use

The detector for this bad smell has identified 15,633 bad smells in Alice (6,362 in attribute and 9,272 in method identifiers) and 820 (133 in attribute and 687 in method identifiers) in WinMerge.

To compute the precision, we randomly selected 20 entries (10 attribute and 10 method identifiers) for each system. The evaluation of the samples indicates a precision of 60% in Alice and 55% in WinMerge. Most of the false positives reported in this case are due to the detector's inability to differentiate the identifiers used to explain (specify) another identifier. For example, a class in WinMerge contains two attributes, named *FWFile* (a file index) and *FWFileName* (a string). *FWFile* is completely contained in *FWFileName*, as it is needed to specify what the name is for. Another type of false positive is represented by the cases when the first identifier represents only part of a word or parts of two consecutive words in the second identifier (e.g., ***Nt*** and *hSi**n**T**h**eta* in Table 3).

Table 3. Sample results of the *inconsistent identifier use* detector in attributes in Alice

Attribute1	Attribute2	Class	Evaluation
Cha	N***cha***nnels	TRDCalROC	Bad smell: cha=chamber
Pids	N***Pids***	AODParticle	Bad smell: NPids=non-zero Pids
HistDcaNegToPrimVertex	***HistDcaNegToPrimVertex*** Zoom	AnalysisTaskStrange	False positive
Nt	hSi***n******T***heta	EMCALJetFinder	False positive

5.4 Useless type indication

The useless type indication detector has identified 121 lexicon bad smells in Alice and 105 in WinMerge. We manually checked all reported bad smells in both systems and found that 115 are correct in Alice (95% precision) and 102 in WinMerge (97% precision).

The false positives are cases when the type of the attribute represents just a part of a word or parts of two consecutive words in the attribute identifier. (e.g., ***int*** in *m_nFlushTime**Int**erval* in Table 4).

Table 4. Sample results of the *useless type indication* detector in attributes in WinMerge

Attribute identifier	Attribute data type	Evaluation
m_nFlushTime***Int***erval	***int***	False positive
m_***ConstraintList***	***ConstraintList***	Bad smell
m_***tempFile***s	***TempFile***	Bad smell
m_***bstr***	***BSTR***	Bad smell

5.5 Identifier construction rules

We ran this detector using a set of identifier construction rules as input, which were defined independently for the two systems, based on the naming conventions adopted in each of them. The detector has located 1,434 violations in Alice (134 in class, 765 in attribute, and 535 in method identifiers) and 1,908 in WinMerge (320 in class, 365 in attribute, and 1,223 in method identifiers).

We manually evaluated the results and found that all the bad smells reported are correct in both systems, leading to a precision of 100%. (See Table 5).

Table 5. Sample results of the *identifier construction rules* detector in classes in Alice

Class Identifier	Violated Rule (regular expression)	Evaluation
SplitGLView	^Ali.*	Bad smell
DrawESD	^Ali.*	Bad smell
ReadPar	^Ali.*	Bad smell
CommandQueue	^Ali.*	Bad smell

6 Discussion

In this study, we evaluated a set of detectors used to identify lexicon bad smells in source code. The evaluation reveals that some of them are very accurate, while others need refinements (see Table 6).

Table 6. Precision of the lexicon bad smell detectors for each system and averages

Lexicon bad smell	Precision (%)		
	Alice	WinMerge	Average
Identifier construction rules	100	100	100
Extremely short terms	80	100	90
Term used to name both the whole and its parts	75-90	75	75-82.5
Inconsistent identifier use	60	55	57.5
Odd grammatical structure	7-10	33-37	20-23.5

The low precision of some of the detectors is due to factors associated with the heuristics used, which sometimes consider overly general cases or they do not take into account common practices in naming. One case of generalization is the *inconsistent identifier use* detector, where all noun specifiers are considered as nouns, while they should be handled differently and noted as specifiers. Also, the *odd grammatical structure* detector uses general grammatical rules and relies on the output of the underlying parser, which

sometimes makes errors. The detector also does not currently recognize commonly accepted coding practices, such as, the use of verbs in boolean attributes. These will be addressed in future versions.

7 Threats to validity

What is considered a bad smell in one system might not be a bad smell in another, due to the specifics of every system. Also, what one person identifies as bad smell might not be interpreted as such by another person. We handle these threats by using configurable files for the particular settings used by our tools and by having the results verified by two and, in some cases, by three researchers and the developers of the system.

The case study can easily be replicated using the detectors and source code of the systems, all available online. However, as the number of some of the reported bad smells was large, we manually evaluated a sample of the result. Even though such a sample was chosen randomly, a different choice might have produced different values of precision.

We did not study the actual impact of the suggested lexicon refactoring on code comprehensibility and quality. We believe that the refactoring would benefit the developers by improving the ease of comprehension, but a study involving human subjects would be needed in order to concur with this.

8 Related work

Researchers have long acknowledged the important role that the lexicon in identifiers and comments plays in understanding and maintaining software systems [1, 2, 4, 6, 8, 12]. Previous work in the field has specifically underlined the importance of high quality, self-explanatory names, and supported the consistent use of the lexicon and of naming rules in source code [4, 5, 9, 11]. The relationship between the use of coding conventions in general and software faults has also been recently studied in [3].

Bad code smells have been introduced by Kent Beck and Martin Fowler in [7] and have been receiving an increasing attention since then [10, 13]. This collection of smells contains also a short list, which refers to the use of lexicon in source code. We refine this list and other bad lexicon smells noted in the literature and include them in our catalog, along with some newly defined bad smells. To our knowledge, this paper is the first attempt to catalog bad smells specific to source code lexicon and to propose detection and refactoring techniques.

9 Conclusions and future work

We introduced the notion of *lexicon bad smell* and presented five of such smells. We developed a series of bad smell detectors and assessed their precision on two large open-source systems. Our study revealed that some bad smells are easy to identify, whereas others are hard to detect and discussed possible improvements in their detection.

Our future work will focus on refining and extending the existing list of bad smells, with the collaboration and feedback of the research community. Also, we plan to extend the definition of the current smells to include more types of identifiers, to implement detectors for new smells and to improve the performance of the current ones. At the same time, we will run case studies on a multitude of systems and observe the differences in the detection and interpretation of bad smells.

10 Acknowledgements

This research has been supported in part by grants from the US National Science Foundation (CCF-0820133 and CCF-0845706).

11 References

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

[2] Biggerstaff, T., Mitbander, B., Webster, D., "The Concept Assignment Problem in Program Understanding", in *Proc. Int. Conf. on Soft. Eng.*, 1994, pp. 482-498.

[3] Boogerd, C. and Moonen, L., "Evaluating the Relation Between Coding Standard Violations and Faults Within and Across Software Versions", in *Proc. Working Conf. on Mining Soft. Repositories*, 2009, pp. 41-50.

[4] Caprile, B. and Tonella, P., "Restructuring Program Identifier Names", in *Proc. Int. Conf. on Soft. Maintenance*, 2000, pp. 97-107.

[5] Deissenboeck, F. and Pizka , M., "Concise and Consistent Naming", *Software Quality Journal*, 14, 3, 2006, pp. 261-282

[6] Etzkorn, L., Bowen, L., Davis, C., "An Approach to Program Understanding by Natural Language Understanding", *Natural Lang. Eng.*, 5(3), 1999, pp.219-236.

[7] Fowler, M., Beck, K., Opdyke, W., Roberts, D., *Refactoring: Improving the Design of Existing Code*, Reading, MA, Addison-Wesley Professional, 1999.

[8] Høst, E., Østvold, B., "The Programmer's Lexicon, Volume I: The Verbs", in *Proc. Int. Working Conf. on Source Code Analysis and Manipulation*, 2007, pp. 193-202.

[9] Lawrie, D., Feild, H., Binkley, D., "Syntactic Identifier Conciseness and Consistency", in *Proc. Workshop on Source Code Analysis and Manipulation*, 2006, pp. 139-148.

[10] Mantyla, M., Bad Smells in Software - a Taxonomy and an Empirical Study, *PhD Thesis*, Helsinki University of Technology, 2003.

[11] Ratiu, D. and Deissenboeck, F., "From Reality to Programs and (Not Quite) Back Again", in *Proc. Int. Conf. on Program Comprehension*, 2007, pp. 91-102.

[12] Takang, A., Grubb, P., Macredie, R., "The Effects of Comments and Identifier Names on Program Comprehensibility: An Experimental Investigation", *Journal of Programming Languages*, 4, 3, 1996, pp. 143-167.

[13] Wake, W., *Refactoring Workbook*, Boston, USA, 2003.