A New Family of Software Anti-Patterns: Linguistic Anti-Patterns

Venera Arnaoudova[1,2], Massimiliano Di Penta[3], Giuliano Antoniol[2], Yann-Gaël Guéhéneuc[1]
[1] *Ptidej Team, DGIGL, École Polytechnique de Montréal, Canada*
[2] *Soccer Lab., DGIGL, École Polytechnique de Montréal, Canada*
[3] *Department of Engineering, University of Sannio, Benevento, Italy*
E-mails: venera.arnaoudova@polymtl.ca, dipenta@unisannio.it, antoniol@ieee.org, yann-gael.gueheneuc@polymtl.ca

***Abstract*—Recent and past studies have shown that poor source code lexicon negatively affects software understandability, maintainability, and, overall, quality. Besides a poor usage of lexicon and documentation, sometimes a software artifact description is misleading with respect to its implementation. Consequently, developers will spend more time and effort when understanding these software artifacts, or even make wrong assumptions when they use them.**

This paper introduces the definition of software linguistic antipatterns, and defines a family of them, i.e., those related to inconsistencies (i) between method signatures, documentation, and behavior and (ii) between attribute names, types, and comments. Whereas "design" antipatterns represent recurring, poor design choices, linguistic antipatterns represent recurring, poor naming and commenting choices.

The paper provides a first catalogue of one family of linguistic antipatterns, showing real examples of such antipatterns and explaining what kind of misunderstanding they can cause. Also, the paper proposes a dectector prototype for Java programs called LAPD (Linguistic Anti-Pattern Detector), and reports a study investigating the presence of linguistic antipatterns in four Java software projects.

***Keywords*-Software antipatterns, Source code lexicon, Textual analysis of software artifacts.**

I. Introduction

Source code lexicon, i.e., the vocabulary used in naming software entities, is an essential element of any software system. A good source code lexicon can positively affect software quality, in particular comprehensibility and maintainability, and even reduce fault-proneness [1], [2], [3].

Several approaches have been developed for better lexicon and coding styles. Some researchers have developed approaches to assess the quality of source code lexicon [2], [4], [5], and some others provided a set of guidelines to produce high-quality identifiers [6].

In summary, existing literature analyzed the quality of source code lexicon solely in terms of what kinds of words were used, e.g., (i) whether identifiers are composed of words belonging to the English dictionary or to a domain specific dictionary; (ii) whether, instead, identifiers contain abbreviations, acronyms, and other combinations of characters. However, sometimes problems in the source code lexicon are more subtle and go beyond the occurrence of words. It may happen that the naming of a method does not properly reflect the method behavior, describing less (or more) than the method actually does. One such example, occurred in Eclipse 1.0, is a method named *isClassPathCorrect* defined in class *ProblemReporter*. One would expect that such a method returns a Boolean; instead, the method does not return any value and sets an attribute and calls another method to perform the task.

This paper represents the starting point for the definition of a new family of software antipatterns, named linguistic antipatterns. Software antipatterns—as they are known so far—are opposite to design patterns [7], i.e., they identify "poor" solutions to recurring design problems, for example, Brown's 40 antipatterns describe the most common pitfalls in the software industry [8]. They are generally introduced by developers not having sufficient knowledge and–or experience in solving a particular problem, or misusing good solutions, i.e., design patterns. Linguistic antipatterns shift the perspective from source code structure towards its consistency with the lexicon:

> **Linguistic Antipatterns (LAs)** in software systems are recurring poor practices in the naming, documentation, and choice of identifiers in the implementation of an entity, thus possibly impairing program understanding.

The presence of inconsistencies can be particularly harmful for developers that can make wrong assumptions about the code behavior or spend unnecessary time and effort to clarify it when understanding source code for their purposes. Therefore, highlighting their presence is essential for producing code easy to understand.

The contributions of this paper are:

1) A first catalogue of a family of LAs, focusing on inconsistencies between method/attribute naming conventions, documentation, and signature. For methods, such LAs are categorized into methods that (i) "do more than they say", (ii) "say more than they do", and (iii) "do the opposite than they say". Similarly, for attributes we categorize the LAs into attributes for which (i) "the name says more than the entity contains", (ii) "the name says less than the entity contains", and (iii) "the name says the opposite than the entity contains".
   For each category, we report different LAs, explaining

how they occur, what kind of misunderstanding they can cause, and reporting examples found in open-source projects. It is important to point out that, since this paper describes LAs for entity names and their documentation, these can occur—and therefore can be detected—in source code, but also in design documents such as class diagrams and API specifications.
2) Possible implementation algorithms—conceived for Java programs—to detect the proposed LAs. The detection is based on a combination of source code structural and textual analysis.
3) A first study reporting occurrence of the described LAs in four Java open-source projects, namely two ArgoUML releases, one release of Cocoon, and one release of Eclipse.

The paper is organized as follows. Section II describes the catalogue of LAs. Section III explains how the LA detection has been implemented. Section IV reports the study conducted on the four software releases, providing both quantitative and qualitative analyses of the detected LAs. Section V discusses the related literature, while Section VI concludes and outlines directions for future work.

II. Linguistic Antipatterns (LAs) Catalogue

An LA is a bad practice involving program entities, their names and/or documentation. One among many possible families of LAs is related to inconsistencies between an entity naming, documentation, and implementation. Within this family, LAs that share important characteristics are grouped into categories. We present six categories of LAs, three of them regarding behavior—i.e., methods—and three regarding state, i.e., attributes. For the definition of each LA we followed the main lines of the software development antipattern template. Specifically, we provide the following information: LA name, description, code element to which it applies (e.g., attribute or method), an example coming from real software projects, and possible consequences of the LA. We leave the possible causes and impact as part of future work, as we believe that they should be inferred by developers and they require an extensive in-field study.

A. Does more than it says

1) "Get" - more than an accessor: In Java, accessor methods, also called getters, provide a way to access class attributes. As such, it is not common that getters perform actions other than returning the corresponding attribute. Any other action should be documented, possibly naming the method differently than "getSomething".

Applies to: Methods

Example: In the example shown in Figure 1, the method *getImageData()*, defined in class *CompositeImageDescriptor*, is not a simple accessor because it never reads the value of the corresponding attribute but it rather sets it with a new object and returns it.

Consequences: The usage of such getters would cause a unexpected allocation of new objects (which normally does not happen with getters), or returning a null value when this should not be the case, i.e., the attribute is not null.

```
public ImageData getImageData() {
  Point size = getSize();
  RGB black = new RGB(0, 0, 0);
  RGB[] rgbs = new RGB[256];
  rgbs[0] = black; // transparency
  rgbs[1] = black; // black
  PaletteData dataPalette=new PaletteData(rgbs);
  imageData = new
    ImageData(size.x,size.y,8,dataPalette);
  imageData.transparentPixel = 0;
  drawCompositeImage(size.x, size.y);
  for (int i = 0; i < rgbs.length; i++)
    if (rgbs[i] == null) rgbs[i] = black;
  return imageData;
}
```

Figure 1. Example of *"Get" - more than an accessor* (Eclipse-1.0).

2) "Is" returns more than a Boolean: When a method name starts with the term "is" one would expect a *Boolean* as a return type, thus having two possible values for the predicate, i.e., "true" and "false". Thus, having an "is" method that does not return a *Boolean*, but returns more information is counterintuitive. In such cases, the method should be renamed or, at least, the fact that it returns a different type should be documented in the method comments.

Applies to: Methods

Example: The example in Figure 2 shows one such case occurring in class *DelayedValidity*. A proper documentation would include details about the return values.

Consequences: Normally, problems related to such LA will be detected at compile time (or even by the IDE), however the misleading naming can still cause misunderstanding from the maintainers' side.

```
public int isValid() {
 final long currentTime =
        System.currentTimeMillis();
 if (currentTime <= this.expires) {
  // The delay has not passed yet -
  // assuming source is valid.
  return SourceValidity.VALID;
 }
 // The delay has passed,
 // prepare for the next interval.
 this.expires = currentTime + this.delay;
 return this.delegate.isValid();
}
```

Figure 2. Example of *"Is" returns more than a Boolean* (Cocoon 2.2.0).

3) "Set" method returns: Modifier methods, i.e., setters, are methods that allow assigning a value to a class attribute (normally protected or private, hence not directly accessible from outside); by convention, setters do not return anything. More generally, the same statement is valid for methods whose name starts with "set". Thus, a set method having a return type different than void should document the return type/values with an appropriate comment or should be named differently to avoid any confusion.

Applies to: Methods
Example: In the example shown in Figure 3 the method *setBreadth*, defined in class *Orientation* returns a *Dimension*.
Consequences: One could use the setter method without storing/checking its returned value, hence useful information—e.g., related to erroneous or unexpected behavior—is not captured.

```
public Dimension setBreadth
        (Dimension target, int source) {
  if (orientation == VERTICAL)
    return new Dimension(source,
        (int)target.getHeight());
  else
    return new Dimension(
                (int)target.getWidth(), source);
}
```

Figure 3. Example of *"Set" method returns* (ArgoUML-0.10.1).

4) Expecting but not getting a single instance: When a method name indicates that a single object (and not a collection) is returned, this shall be consistent with its return type. If, instead, the return type is a collection, the method shall be renamed or appropriate documentation is needed.
Applies to: Methods
Example: The name of the method in Figure 4—defined in class *DrillFrame*—suggests that an object *Expansion* will be returned whereas a collection is (the return type is *List*). Very likely, the reader would not know what types of objects are contained in this list.
Consequences: Although this would unlikely cause faults at run-time, it might cause false expectancies to the developers. When reading "returnPath", one would expect to handle a simple object, whereas it is necessary to deal with multiple objects.

```
/* Returns the expansion state for a tree.
 * @return the expansion state for a tree */
public List getExpansion() { return fExpansion;}
```

Figure 4. Example of *Expecting but not getting a single instance* (Eclipse-1.0).

B. Says more than it does

1) Not implemented condition: Comments suggest a conditional behavior, while the code does not.
Applies to: Methods
Example: Figure 5 shows a method defined in class *FileEditionEditorInput* that always returns the same value.
Consequences: This LA can have two main consequences. First, clients of the corresponding methods assume the documented behavior resulting in wrong system behavior. Second, during testing—especially black box testing—the tester would invest time and effort to generate test cases for the different conditions, while one test case will cover all method statements (or, in general, less test cases are needed).

2) Validation method does not confirm: A validation method does not return a value to confirm such validation.
Applies to: Methods

```
/* Returns the children of this object.
 * When this object is displayed in a tree,
 * the returned objects will be this element's
 * children. Returns an empty array if this
 * object has no children.
 * @param object The object to get the
 * children for. */
public Object[] getChildren(Object o)
 { return new Object[0]; }
```

Figure 5. Example of *Not implemented condition* (Eclipse-1.0).

Example: Figure 6 shows a method defined in class *UMLComboBoxEntry* that neither returns a Boolean nor throws an exception.
Consequences: One may not know how to handle the outcome of the validation. Very likely, such an outcome is stored somewhere—e.g., an instance variable—however this is not clear from the method specification/documentation.

```
public void checkCollision(String before,
        String after){
boolean collision=(before!=null
   && before.equals(_shortName)) ||
   (after!=null&&after.equals(_shortName));
 if(collision) {
  if(_longName==null){_longName=getLongName();}
  _displayName = _longName;
 }
}
```

Figure 6. Example of *Validation method does not confirm* (ArgoUML-0.10.1).

3) "Get" method does not return: The name suggests that the method returns something, however this is not the case.
Applies to: Methods
Example: The example in Figure 7 shows the source code of a method named *getMethodBodies*, defined in class *Compiler*, which suggests a field as result, however nothing is returned.
Consequences: One would expect to be able to assign the method return value to a variable. However, since this is not possible, one has to further understand the code to determine where the retrieved data is stored and how to obtain it.

4) Not answered question: The method name is in the form of predicate whereas the return type is not Boolean.
Applies to: Methods
Example: Figure 8 shows an example of a method, declared in class *ISelectionValidator*, where the name suggests a Boolean value as result but nothing is returned.
Consequences: Consequences are similar to those of *"Get" method does not return*. In this case, the developer would even expect to use the method within a conditional control structure, which is however not possible.

5) Transform method does not return: The method name suggests the transformation of an object, but there is no return value.

```
protected void getMethodBodies
  (CompilationUnitDeclaration unit,int place){
  //fill the methods bodies in order
  //for the code to be generated
  if (unit.ignoreMethodBodies) {
    unit.ignoreFurtherInvestigation = true;
    return; // if initial diet parse did not
    //work, no need to dig into method bodies.
  }
  if (place < parseThreshold)
    return; //work already done ...
  //real parse of the method....
  parser.scanner.setSourceBuffer(
     unit.compilationResult.
     compilationUnit.getContents());
  if (unit.types != null) {
    for (int i = unit.types.length; --i >= 0;)
      unit.types[i].parseMethod(parser, unit);
        }
}
```

Figure 7. Example of *"Get" method does not return* (Eclipse-1.0).

```
public void isValid(
        Object[] selection, StatusInfo res) {
  // only single selection
  if (selection.length == 1
          && (selection[0] instanceof IFile))
    res.setOK();
  else res.setError(""); //$NON-NLS-1$
}
```

Figure 8. Example of *Not answered question* (Eclipse-1.0).

Applies to: Methods
Example: An example of this LA is method *javaToNative* defined in class *LocalSelectionTransfer* shown in Figure 9.
Consequences: Similar to the previous ones. Specifically, here one would expect to be able to assign the result of the method to a variable suggested by the method name (*Native* in our example, i.e., a platform-specific representation).

```
public void javaToNative(
    Object object, TransferData transferData) {
  byte[] check= TYPE_NAME.getBytes();
  super.javaToNative(check, transferData);
}
```

Figure 9. Example of *Transform method does not return* (Eclipse-1.0).

6) Expecting but not getting a collection: The method name suggests that a collection should be returned, however a single object, or nothing, is returned.
Applies to: Methods
Example: In the example shown in Figure 10, the name of the method, defined in class *SAXParserBase*, suggests that some statistics will be returned, while the method only returns a Boolean value.
Consequences: A developer would likely expect that the method will return a set of values (e.g., a time series of temperature, or an array of monitoring data), suggesting that appropriate patterns, such as iterators, are needed to navigate the data structure. Instead, in some cases, the method may return only one of these values, or, in other cases, like the one in Figure 10, the returned value is completely inconsistent with the method name.

```
public boolean getStats(){ return _stats; }
```

Figure 10. Example of *Expecting but not getting a collection* (ArgoUML-0.10.1).

C. Does the opposite

1) Method name and return type are opposite: The intent of the method suggested by its name is in contradiction with what it returns.
Applies to: Methods
Example: The method shown in Figure 11, defined in class *ControlEnableState*, is an example of this LA, where the name and return type are inconsistent because the method *disable* returns an "enable" state. With the available documentation, the reader will infer that the return type is a control state that can be enabled or disabled.
Consequences: The developers can make wrong assumptions on the returned value and this might not be discovered at compile time. In some cases—e.g., when the method returns a Boolean—the developer could negate (or not) the value where it should not be negated (or it should be).

```
/* Saves the current enable/disable state of
 * the given control and its descendents in the
 * returned object;the controls are all disabled.
 * @param w the control
 * @return an object capturing the enable/disable
 * state */
public static ControlEnableState
    disable(Control w){
       return new ControlEnableState(w);
}
```

Figure 11. Example of *Method name and return type are opposite* (Eclipse-1.0).

2) Method signature and comment are opposite: The documentation of a method is in contradiction with its declaration (e.g., name, return type).
Applies to: Methods
Example: Figure 12 shows a method, declared in class *NavigationHistory*, which name suggests that the method returns true if the forward navigation is enabled. However, the comment talks about back navigation, i.e., the opposite.
Consequences: Consequences are similar to those of the *Method name and return type are opposite*, and can be even more misleading because the developer is unsure whether to trust the comment or the method signature. Either the one or the other is outdated or inconsistent, and has to be updated.

D. Contains more than it says

1) Says one but contains many: An attribute name suggests a single instance, while its type suggests that the attribute stores a collection of objects.
Applies to: Attributes
Example: Figure 13 shows an attribute, defined in class *TableModelCritics*, which name suggests a single object, whereas its type is a collection.

```
/* Returns true if this listener has a target
 * for a back navigation.  Only one listener
 * needs to return true for the back button
 * to be enabled. */
public boolean isNavigateForwardEnabled() {
  boolean enabled = false;
  if(_isForwardEnabled==1) {enabled = true;}
  else {
    if(_isForwardEnabled != 0) {
      enabled = navigateForward(false) != null;
    }
  }
  return enabled;
}
```

Figure 12. Example of *Method signature and comment are opposite* (ArgoUML-0.10.1).

Consequences: Lack of understanding of the class state/associations. When such attribute changes, one would not know whether the change impacts a one or multiple objects.

```
Vector _target;
```

Figure 13. Example of *Says one but contains many* (ArgoUML-0.10.1).

2) Name suggests Boolean but type does not: An attribute name suggests that its value is true or false, but its declaring type is not *Boolean*.
Applies to: Attributes
Example: Figure 14 shows one such case defined in class *ExceptionHandlingFlowContext*. The attribute name suggests that the value will be true if something is reached, false otherwise. However, the declaring type is not Boolean.
Consequence: The developer would expect to be able to test the attribute in a control flow statement condition. However, this is not the case, especially in cases like the one in Figure 14, for which the returned type is an array, therefore it is not clear how to handle this attribute.

```
int[] isReached;
```

Figure 14. Example of *Name suggests Boolean but type does not* (Eclipse-1.0).

E. Says more than it contains

1) Says many but contains one: An attribute name suggests multiple instances, but its type suggests a single one.
Applies to: Attributes
Example: In the example shown in Figure 15, the attribute name, defined in class *SAXParserBase*, suggests that it contains statistics whereas its type is *Boolean*.
Consequences: Lack of understanding of the impact of attribute changes (see also *Says one but contains many*).

```
private static boolean _stats = true;
```

Figure 15. Example of *Says many but contains one* (ArgoUML-0.10.1).

F. Contains the opposite

1) Attribute name and type are opposite: The name of an attribute is in contradiction with its type.
Applies to: Attributes
Example: The example of Figure 16 shows an attribute of class *ActionNavigability*. The contradiction comes form the use of the antonyms *start* and *end*, one being part of the type of the attribute, the other being part of its name.
Consequences: This kind of misleading attribute naming can induce wrong assumptions. For example, whether a *Boolean* attribute contains information that can be used directly in a control flow statement condition, or whether it has to be negated. Similarly, prefixes/suffixes such as "start" and "end" could confuse the developer about the direction a data structure should be traversed.

```
MAssociationEnd start = null;
```

Figure 16. Example of *Attribute name and type are opposite* (ArgoUML-0.10.1).

2) Attribute signature and comment are opposite: The documentation of the entity is in contradiction with its declaration (e.g., name, type).
Applies to: Attributes
Example: The example in Figure 17 shows an attribute named *INCLUDE_NAME_DEFAULT*, defined in class *EncodeURLTransformer*. However, its comment says *"Configuration default exclude pattern"*. Whether the pattern is included or excluded is therefore unclear from the comment and name.
Consequences Without a deep analysis of the source code, the developer might not clearly understand the role of the attribute, and the comment is just misleading.

```
/* Configuration default exclude pattern,
 * ie .*\/@href|.*\/@action|frame/@src */
public final static String INCLUDE_NAME_DEFAULT
= ".*/@href=|.*/@action=|frame/@src=";
```

Figure 17. Example of *Attribute signature and comment are opposite* (Cocoon-2.2.0).

III. Tool Support

This section describes possible detection algorithms for the LAs described in Section II, and the technology we used to implement those algorithms.

"Get" - more than an accessor (Section II-A1) Find accessor methods (i.e., the method name starts with "get" and ends with a substring that corresponds to an attribute in the same class) and identify those that are performing more actions than returning the corresponding attribute (e.g., using measures such as LOC or McCabe's Cyclomatic Complexity). Cases where the attribute is set before it is returned (i.e., Proxy and Singleton design patterns) should not be considered as part of this LA.
"Is" returns more than a Boolean (Section II-A2) Find methods starting with "is" whose return type is not *Boolean*.
"Set" method returns (Section II-A3) Find modifier methods (or more generally methods whose name starts with "set") and whose return type is different from *void*.
Expecting but not getting a single instance (Section II-A4) Find methods returning a collection (e.g., array, list, vector,

etc.) but whose name ends with a singular noun.
Not implemented condition (Section II-B1) Find methods with at least one conditional sentence in comments but with no conditional statements in the implementation (e.g., no control structures or ternary operators).
Validation method does not confirm (Section II-B2) Find validation methods (e.g., method names starting with "validate", "check", "ensure") whose return type is *void* and that do not throw exceptions.
"Get" method does not return (Section II-B3) Find methods whose names suggest a return value (e.g., names starting with "get", "return") and whose return type is *void*.
Not answered question (Section II-B4) Find methods whose name is in the form of predicate (e.g., starts with "is", "has") and whose return type is *void*.
Transform method does not return (Section II-B5) Find methods whose name suggests a data transformation (e.g., *toSomething*, *source2target*) but its return type is *void*.
Expecting but not getting a collection (Section II-B6) The method name suggests that it returns (e.g., starts with "get", "return") multiple objects (e.g., ends with a plural noun), however the return type is not a collection.
Method name and return type are opposite (Section II-C1) Find methods with antonyms in name and return type.
Method signature and comment are opposite (Section II-C2) Find methods whose name or return type have an antonym relation with its comment.
Says one but contains many (Section II-D1) Find attributes having a name ending with a singular noun, as well as a collection as return type.
Name suggests Boolean but type does not (Section II-D2) Find attributes whose name is structured as a predicate i.e., starting with a verb in third person (e.g., "is", "has") or ending with a verb in gerund, or present participle, but whose declaring type is not *Boolean*.
Says many but contains one (Section II-E1) Find attributes having a name ending with a plural noun, however their type is not a collection neither it contains a plural noun.
Attribute name and type are opposite (Section II-F1) Find attributes whose name and declaring type contain antonyms.
Attribute signature and comment are opposite (Section II-F2) Find attributes whose name or declaring type have an antonym relation with its comment.

In the subsequent paragraphs we provide details on the technology we used to implement the detection. The source code analysis consists in three parts: (i) fact extraction from source code, (ii) analysis of source code identifiers and comments, and (iii) establishing semantic relations between terms contained in identifiers and comments.

Fact extraction from source code. To this aim, we use the *srcml* tool [9], which parses source code and produces an XML-based parse tree. With this step, we identify the various source code elements of interest for our analysis, namely attribute names and types, method names, return types, parameter names and types, exceptions. In addition, we can also extract from source code other pieces of information needed for our analysis, i.e., the presence of control flow or conditional statement, the usage of particular variables/parameters in conditional statements, and exception handling.

The fact extractor also identifies comments in the source code. In general, a comment is attached to the entity it precedes. However, when a comment follows an entity declaration and it starts at the same line then it is attached to the preceding entity.

Analysis of source code identifiers and comments. This step aims at identifying term composing identifiers, and performing a part of speech analysis. First, identifiers are split using the camel case and underscore heuristics. For Java, this is largely sufficient [10].

After having extracted terms, we perform a part of speech analysis using the *Stanford* natural language parser [11]. This allows to: (i) identify whether a term is a noun, an adjective, an adverb or other parts-of-speech; (ii) distinguish singular from plural nouns; and (iii) identify dependencies between words, e.g., between subjects and predicates, as well as negative forms, e.g., *not possible*.

Relating terms occurring in source code and comments. The last part of the analysis aims at relating terms appearing in various source code elements and comments, e.g., synonymy and antonymy relations. Such relations are established using the WordNet ontology [12]. Although we are aware that WordNet may not necessarily be the most suitable ontology to analyze source code, as pointed out by Hindle *et al.* [13]—at the moment WordNet represents, to the best of our knowledge, the most suitable technology for this task. Also, the approach is perfectly applicable even if replacing WordNet with a domain-specific ontology.

IV. Study Description

The *goal* of this study is to investigate the presence of LAs in software systems, with the *purpose* of understanding the relevance of the phenomenon. The *quality focus* is software comprehensibility that can be hindered by LAs. The *perspective* is of researchers interested to develop recommending systems aimed at detecting the presence of LAs and suggesting ways to avoid them. The *context* consists of four Java systems, namely two versions of *ArgoUML*, one version of Cocoon, and one version of Eclipse. Table I reports some information about the systems, namely number of lines of code and comments, number of classes, methods, and attributes. *ArgoUML*[1] is an UML modeller and reverse engineering tool. *Cocoon*[2] is a framework based on Spring that allows to develop Web applications by integrating components into pipelines. *Eclipse*[3] is a well known framework

[1]http://argouml.tigris.org
[2]http://cocoon.apache.org
[3]http://www.eclipse.org

Table I
ANALYZED SYSTEMS.

System	Vers.	Code	Comm.	Classes	Meth.	Attr.
Argo-UML	0.10.1	82K	40K	896	5 363	2 986
	0.34	195K	150K	2 426	10 876	6 102
Cocoon	2.2.0	60K	43K	748	3 947	2 717
Eclipse	1.0	475K	248K	6 388	35 508	21 985

and IDE. We have chosen systems having different size, and for one of them both an old version and a new one.

The study aims at answering the research question:

> **RQ:** To what extent do the analyzed systems contain the LAs defined in Section II?

In Section IV-A we describe the qualitative study, in Section IV-B we discuss the precision of the approach, whereas in Section IV-C we analyze LAs over time. It is important to point out that, rather than a large, quantitative study, this is more a qualitative investigation. Other than reporting the number of LAs each program contains, we will discuss in detail some examples to better understand the nature of the phenomenon.

A. Study Results

Table II reports, for each system, the number of detected and validated LAs, as well as the precision of the implemented algorithms. The validated sample for each LA is randomly selected and its size is statistically significant considering a confidence level of 95% and a confidence interval of $\pm 10\%$ [14].

"Get" - more than an accessor (Section II-A1): Interesting examples of this LA are cases where the method performs some calculations and returns the result. This is the case of *getCurrentCommentOffset*, defined in class *CodeFormatter* (Eclipse), which computes and returns the number of characters and tabs between the beginning of a line and the beginning of a comment.

"Is" returns more than a Boolean (Section II-A2): In all of the detected cases of this LA the return type is either *int* or *String*. In the first case, the integer values distinguish different situations. One may argue that *int* is used instead of *Boolean* to encode logical values, as it happens in C, i.e., 0 corresponds to false and 1 to true. However, this is not the case in the analyzed LAs. For example, the method shown in Figure 2 has three possible return values: -1 (which corresponds to "invalid"), 1 ("valid"), and 0 ("don't know").

"Set" method returns (Section II-A3). Detected cases of this LA share several common characteristics. When the return value is *Boolean*, it often indicates whether the assignment has been successful or not. Some implementations create an object—possibly using the received parameters—assign it to an attribute, and finally return it. Sometimes, as in the example of Figure 3, no assignment is performed, and a new object is created and returned. The example of method *setResponseHandler* declared in class *Client* (Eclipse) is even more counterintuitive, as it returns the old value, or null if no old value existed.

Expecting but not getting a single instance (Section II-A4): Examples of this LA are cases where the returned object represents a data structure that is not encapsulated in a separate class, such as method *getObjectModel* of class *AbstractEnvironment*, returning a *Map* (Cocoon), and method *getCriticRegistry* of class *Agency*, returning a *Hashtable* (ArgoUML-0.10.1).

Not implemented condition (Section II-B1): Examples of this LA can be grouped into those returning always the same value. For example, method *loadUnspecified* from class *ConfigurationHandler* (ArgoUML-0.10.1) is documented with the comment *"@return true if the load was successful, otherwise false."*. However, the method always returns *false*. The same LA exists in version 0.34. There are also cases where the implementation is temporary, as method *getValue* of class *EvaluationResult* (Eclipse) where the body always returns *null* accompanied with the comment *"Not yet implemented"*).

Validation method does not confirm (Section II-B2): Examples of this LA are methods that, other than checking the condition, also perform some action. One example is method *checkJVMVersion* of class *Main* (ArgoUML-0.34), which checks if the JVM is not supported, and, in that case, terminates the program. Another one is method *ensureFirstCharLowerCase* of class *FormatingStrategyUML*, which implementation returns a new object following the specification, rather than ensuring if the parameter follows the specification. Other examples are those where the result is stored in an instance variable as in method *checkForEncoding* of class *EncodeURLTransformer* (Cocoon).

"Get" method does not return (Section II-B3. We observe two main practices when this LA occurs. The first one occurs when the result is assigned to instance variable(s), as in method *getPreferences* of class *VirtualMachineManagerImpl* (Eclipse), where *loadPreferences* would better reflects the functionality as the comment also suggests *"Loads the user preferences from the jdi.ini file"*. The second one occurs when parameters are modified rather than returning a value, as in method *getTypeQualifiedName* of class *JavaModelUtility*, where the qualified name of the type (first parameter) is stored in a buffer (second parameter).

Not answered question (Section II-B4). We found examples, in which methods are modifiers for a *Boolean* attribute, e.g., method *isSelected* of class *ComponentEntryDescriptor* (Eclipse). Others throw exceptions instead of returning a *Boolean* value, as method *isNotNull* of class *Assert* (Eclipse). Finally, there are cases of methods that perform some undocumented actions, as method *isClassPathCorrect* of class *ProblemReporter* (Eclipse), where an attribute is set after which the method *handle* is invoked. The latter in turn delegates to its parent, finally throwing a runtime exception in case there is no reference context.

Table II
DETECTED LAS

	ArgoUML 0.10.1	ArgoUML 0.34	Cocoon 2.2.0	Eclipse 1.0	Validated	TP	Precision
"Get" - more than an accessor (Section II-A1)	0	2	1	15	18/18	12	67%
"Is" returns more than a Boolean (Section II-A2)	2	0	4	26	24/32	24	100%
"Set" method returns (Section II-A3)	4	30	6	53	47/93	46	98%
Expecting but not getting a single instance (Section II-A4)	7	3	8	33	34/51	26	77%
Not implemented condition (Section II-B1)	20	28	43	232	74/323	58	78 %
Validation method does not confirm (Section II-B2)	1	8	11	235	70/255	52	74%
"Get" method does not return (Section II-B3)	1	3	2	57	38/63	37	97%
Not answered question (Section II-B4)	0	2	0	34	36/36	36	100%
Transform method does not return (Section II-B5)	0	86	15	44	59/145	58	98%
Expecting but not getting a collection (Section II-B6)	8	39	12	135	66/194	49	74%
Method name and return type are opposite (Section II-C1)	0	0	0	6	6/6	3	50%
Method signature and comment are opposite (Section II-C2)	7	20	12	243	72/282	6	8%
Says one but contains many (Section II-D1)	15	92	42	103	70/252	40	57%
Name suggests Boolean but type does not (Section II-D2)	14	13	21	138	64/186	36	56%
Says many but contains one (Section II-E1)	45	117	24	116	73/302	55	75%
Attribute name and type are opposite (Section II-F1)	1	0	0	0	1/1	1	100%
Attribute signature and comment are opposite (Section II-F2)	1	0	3	19	23/23	2	9%

Transform method does not return (Section II-B5). A common characteristic in the examples of this LA is that the expected return value is assigned to a parameter rather than being returned, e.g., method *readerToWriter* of class *ZargoFilePersister* (ArgoUML-0.34), method *ToSource* of class *IdentifierExpression* (ArgoUML-0.34), and method *toString* of class *Parameter* (a member class in Cocoon). Also, there are cases of methods with no parameters, modifying an attribute rather than returning a value, e.g., method *d2f* (double to float) of class *CodeStream* (Eclipse).

Expecting but not getting a collection (Section II-B6). Concerning this LA, we found cases, in which the source code performs some kind of aggregation over multiple objects and returns the result. One example is method *getRows* of class *ClassdiagramLayouter* (ArgoUML-0.10.1) which returns the number of rows of a table. Another is method *getTabs* of class *StyledText* (Eclipse) where, as the comment suggests, the method returns the *"tab width measured in characters"*. In other cases, the parameter is modified and nothing is returned, as in method *getEdges* of class *PgmlUtility* (ArgoUML-0.34), where the method iterates over a collection of edges (passed as parameter) and adds them to another collection of edges (again passed as parameter) if they are of specific type.

Says one but contains many (Section II-D1). Examples from this LA include attribute *methodIndex* of type *HashMap* defined in class *AbstractMultiAction*, attribute *objectModel* of type *Map* defined in class *XModuleSource*, attribute *uploadStatus* of type *Hashtable* defined in class *MultipartParser* (all three from Cocoon), attribute *fExpansion* of type *List* defined in class DrillFrame (Eclipse). Such cases can be easily fixed by renaming the attributes, e.g., *uploadStatus* → *uploadStatuses*.

Name suggests Boolean but type does not (Section II-D2). In the examples we found, we can distinguish cases where the attribute type is *int* and its value corresponds to a specific state. Examples include *IsArrayType* (class *TagBits* of Eclipse), which value is *0x0001*, and DONE_SAVING (class *SaveManager* of Eclipse), with value *3*. For these attributes, having an appropriate documentation is crucial. Other examples include attributes of type *String*, such as *KEY_MISSING* (class *ComponentSpecPage* of Eclipse) assigned to value "NewComponentWizard.SpecPage.missing", *HAS_EXPIRED_NO* (class *WebContinuationDataBean* of Cocoon) assigned to value "no". For those examples, a renaming is recommended. Finally, other examples from various types are attributes *requiresParent* of type *Composite*, defined in class *RequiresSection* (Cocoon), and *needsCompileList* of type *Vector* defined in class *WorkQueue* (Eclipse).

Method name and return type are opposite (Section II-C1). All detected examples of this LA are from Eclipse. Other than the example shown in Figure 11, we found method *implicitSuperConstructorCall* of class *SuperReference*, which return type is *ExplicitConstructorCall* where the words "explicit" and "implicit" are antonyms. In this example a clarifying documentation would ease the comprehension process. The other three cases of detected LAs are false positives as the opposite words are "query" (in method name) and "result" (in the method return type) which does not bring contradiction. One such example is method *queryEntriesMatching* defined in class *BlocksIndexInput* (Eclipse), returning *IEntryResult*.

Method signature and comment are opposite (Section II-C2). Examples of this LA include method *remove*, defined in class *AdaptableList* (Eclipse), where the comment is *"Adds the given adaptable object to this list"*, and method *endTransformingElement*, defined in class *AbstractSAXTransformer* (Cocoon), where the comment begins as follows: *"Start processing elements of our namespace"*.

Says many but contains one (Section II-E1). Examples

include attribute *METHODS* of type *int* declared in *LexicalSorter* (Eclipse) whose value is 5 represents a numeric encoding for the type of an element (5 for method, 6 for static fields, etc.) and attribute *EXAMPLES* of type *String* is the defined in interface *IDocumentSection*, where the value corresponds to a section identifier.

Attribute name and type are opposite (Section II-F1). The only detected example of this LA is the one shown in Figure 16. In essence, this kind of LA is less likely to occur than others, at least based on our preliminary analysis.

Attribute signature and comment are opposite (Section II-F2). Except the example shown in Figure 17, the only other example from the validated sample is attribute *RESULT_DOCID_ATTR*, defined in class *XPathTraversableGenerator* (Cocoon), with comment *"The document containing a successful XPath query"*, where the words "result" and "query" are antonyms.

B. Detection Performance

Based on the validated sample, LAPD has an average precision of 72% (see Table II). There are two cases in which the precision is below 10% and those are *Attribute signature and comment are opposite* and *Method signature and comment are opposite*. This is due to the difficulty of capturing opposite meaning. An example of the latter is method *close*, defined in class *DeltaProcessor* (Eclipse), with comment *"Closes the given element, which removes it from the cache of open elements"*, which will be detected because of the antonyms "open" and "close".

C. LAs across multiple releases of ArgoUML

The 126 LAs detected in ArgoUML-0.10.1 occurred in 81 Java files from which 29 exist (with the same name) also in ArgoUML-0.34, being it in the same package or in a different one. From the 44 LAs detected in those files, 7 were discarded because they were false positives. From the remaining 37, 8 were true positives, one of which was removed and the other 7 remained unchanged in ArgoUML-0.34. The LA that was removed was *Says many but contains one* (Section II-E1) where the attribute *PROPERTIES*, of type *String* was renamed to *propertyLocation*.

D. Threats to Validity

Being this a qualitative analysis, we mainly discuss threats to construct and external validity, while other threats such as internal and conclusion validity do not apply.

Threats to *construct validity* concern the relationship between the theory and the observation, and in this work are mainly due to the mapping between the LA definitions and their detection procedure. In terms of precision, we have mitigated such a threat by manually analyzing a sample of the detected LAs. However, we are aware that the provided LA implementation may suffer of *false negative* problems, i.e., may not capture all actual LAs. As said, one limitations of the current detection is the used WordNet ontology, which might not be fully suitable to analyze software artifacts [13].

Threats to *external validity* concern the generalization of our findings. As said above, this is merely a qualitative analysis aimed at illustrating examples of LAs, rather than at empirically characterizing the phenomenon.

V. Related Work

Several authors, such as Caprile and Tonella [15], Caprile and Merlo [4], [5], and Anquetil *et al.* [16] conducted studies on the structure and informativeness of source code identifiers. They found that identifiers are one of the most important source of information about system concepts, and that the information carried by identifiers is often the starting point for program comprehension. Abebe *et al.* [17] showed that lexicon bad smells—identified in terms of excessive use of contractions or odd grammatical structure—can cause problems during software maintenance, specifically for what concerns concept location. Last, but not least, Abebe *et al.* [18] showed that lexicon bad smells improve fault prediction. While those works focus on smells related to single identifiers, the LAs described in this paper shift the perspective to a higher level of details, considering inconsistencies between method names, parameters, return types and comments, and inconsistencies between attribute names, types, and comments.

Takang *et al.* [1] investigated the role played by identifiers and comments on source code comprehensibility. Their study showed that commented programs are more understandable than non-commented ones and that programs containing full-word identifiers are more understandable than those with abbreviated identifiers. Lawrie *et al.* [2], [3] performed an empirical study to assess the quality of source code identifiers. Their study results suggest that the identification of words composing identifiers, and, thus, of the domain concepts associated with them, could contribute to a better comprehension.

Some authors proposed approaches aimed at supporting the improvement of source code lexicon. Deißenböck and Pizka [6] provided guidelines for the production of high-quality identifiers. Corbo *et al.* proposed SmartFormatter [19], a tool able to learn coding styles from an existing code base, and recommend new developers about indentation styles, usage of comments, and naming conventions used for different kinds of identifiers. Another approach for recommending coding styles was proposed by Reiss [20]. De Lucia *et al.* proposed COCONUT [21], a tool highlighting the (lack of) consistency between terms in requirements and related source code artifacts. Tan *et al.* [22] proposed @tComment to detect inconsistencies between Javadoc and implementation with respect to null values and exceptions. Again, LAPD is complementary to these approaches, as it deals with inconsistencies between different parts of the source code (e.g., method comments versus it's name)

as opposed to COCONUT, and is domain independent as opposed to @TCOMMENT.

VI. Conclusion and Future Work

This paper introduces the concept of Linguistic Antipatterns (LAs), and details one family of LAs intended as recurring inconsistencies in method name, signature, and comments, as well as in attribute name, signature, and comments. The paper reports a catalogue of LAs concerning:

- *methods*, categorized in cases where a method (i) does more than it says, (ii) says more than it does, and (iii) does the opposite than it says.
- *attributes*, categorized in cases where an attribute (i) contains more than it says (ii) says more than it contains, and (iii) contains the opposite than it says.

The catalogue provides examples of LAs from real systems, illustrated their possible consequences, and outlines possible strategies for their detection.

We have carried out a study investigating the presence of LAs in four Java systems, i.e., two ArgoUML releases, one release of Cocoon, and one release of Eclipse. The study is based on a first implementation of detector, named Linguistic AntiPattern Detector (LAPD), with a precision of 72%.

Future work aims at investigating a wider set of LAs, at enhancing the detection approach with further heuristics, and performing an extensive empirical study on the presence of LAs in a large number of software projects, developed in languages other than Java. We also plan to survey developers on possible causes and impact of LAs.

References

[1] A. Takang, P. A. Grubb, and R. D. Macredie, "The effects of comments and identifier names on program comprehensibility: an experiential study," *Journal of Program Languages*, vol. 4, no. 3, pp. 143–167, 1996.

[2] D. Lawrie, C. Morrell, H. Feild, and D. Binkley, "Effective identifier names for comprehension and memory," *Innovations in Systems and Software Engineering*, vol. 3, no. 4, pp. 303–318, 2007.

[3] ——, "What's in a name? a study of identifiers," in *Proceedings of the International Conference on Program Comprehension (ICPC)*. IEEE CS Press, 2006, pp. 3–12.

[4] B. Caprile and P. Tonella, "Restructuring program identifier names," in *Proceedings of the International Conference on Software Maintenance (ICSM)*. IEEE CS Press, 2000, pp. 97–107.

[5] E. Merlo, I. McAdam, and R. De Mori, "Feed-forward and recurrent neural networks for source code informal information analysis," *Journal of Software Maintenance*, vol. 15, no. 4, pp. 205–244, 2003.

[6] F. Deissenbock and M. Pizka, "Concise and consistent naming," in *Proceedings of the International Workshop on Program Comprehension (IWPC)*. IEEE CS Press, 2005.

[7] E. Gamma, R. Helm, R.Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object Oriented Software*. Boston, MA, USA: Addison-Wesley, 1995.

[8] W. J. Brown, R. C. Malveau, H. W. M. III, and T. J. Mowbray, *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*, T. Hudson, Ed. John Wiley & Sons, Inc., 1998.

[9] M. Collard, H. Kagdi, and J. Maletic, "An XML-based lightweight C++ fact extractor," in *Proceedings of the International Workshop on Program Comprehension (IWPC)*. IEEE CS Press, 2003, pp. 134–143.

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

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

[12] G. A. Miller, "WordNet: A lexical database for English," *Communications of the ACM*, vol. 38, no. 11, pp. 39–41, 1995.

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

[14] D. J. Sheskin, *Handbook of Parametric and Nonparametric Statistical Procedures*. Chapman & All, 2007.

[15] B. Caprile and P. Tonella, "Nomen est omen: Analyzing the language of function identifiers," in *Proceedings of Working Conference on Reverse Engineering (WCRE)*, 1999, pp. 112–122.

[16] N. Anquetil and T. Lethbridge, "Assessing the relevance of identifier names in a legacy software system," in *Proceedings of CASCON*, 1998, pp. 213–222.

[17] S. L. Abebe, S. Haiduc, P. Tonella, and A. Marcus, "Lexicon bad smells in software," in *Proceedings of the Working Conference on Reverse Engineering (WCRE)*. IEEE CS Press, 2009, pp. 95–99.

[18] S. L. Abebe, V. Arnaoudova, P. Tonella, G. Antoniol, and Y.-G. Guéhéneuc, "Can lexicon bad smells improve fault prediction?" in *Proceedings of the Working Conference on Reverse Engineering (WCRE)*, 2012, pp. 235–244.

[19] F. Corbo, C. Del Grosso, and M. Di Penta, "Smart formatter: Learning coding style from existing source code," in *Proceedings of the International Conference on Software Maintenance (ICSM)*, 2007, pp. 525–526.

[20] S. P. Reiss, "Automatic code stylizing," in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2007, pp. 74–83.

[21] A. De Lucia, M. Di Penta, and R. Oliveto, "Improving source code lexicon via traceability and information retrieval," *IEEE Transactions on Software Engineering*, vol. 37, no. 2, pp. 205–227, 2011.

[22] S. H. Tan, D. Marinov, L. Tan, and G. T. Leavens, "@tComment: Testing Javadoc comments to detect comment-code inconsistencies," in *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*, 2012, pp. 260–269.