

See discussions, stats, and author profiles for this publication at: <https://www.researchgate.net/publication/276314133>

Linguistic antipatterns: what they are and how developers perceive them

Article in Empirical Software Engineering · January 2015

DOI: 10.1007/s10664-014-9350-8

CITATIONS

74

READS

2,882

3 authors:



Venera Arnaoudova

Washington State University

35 PUBLICATIONS 538 CITATIONS

SEE PROFILE



Massimiliano Di Penta

Università degli Studi del Sannio

383 PUBLICATIONS 15,048 CITATIONS

SEE PROFILE



Giuliano Antoniol

Polytechnique Montréal

327 PUBLICATIONS 10,680 CITATIONS

SEE PROFILE

Some of the authors of this publication are also working on these related projects:



AUTOMATIC EVALUATION AND IMPROVEMENT OF SOFTWARE ARCHITECTURE [View project](#)



Estimating the Number of Remaining Links in Traceability Recovery [View project](#)

Linguistic Antipatterns: What They Are and How Developers Perceive Them

Venera Arnaoudova · Massimiliano Di Penta · Giuliano Antoniol

Received: date / Accepted: date

Abstract Antipatterns are known as poor solutions to recurring problems. For example, Brown *et al.* and Fowler define practices concerning poor design or implementation solutions. However, we know that the source code lexicon is part of the factors that affect the psychological complexity of a program, i.e., factors that make a program difficult to understand and maintain by humans. The aim of this work is to identify recurring poor practices related to inconsistencies among the naming, documentation, and implementation of an entity—called Linguistic Antipatterns (LAs)—that may impair program understanding. To this end, we first mine examples of such inconsistencies in real open-source projects and abstract them into a catalog of 17 recurring LAs related to methods and attributes¹. Then, to understand the relevancy of LAs, we perform two empirical studies with developers—30 *external* (i.e., not familiar with the code) and 14 *internal* (i.e., people developing or maintaining the code). Results indicate that the majority of the participants perceive LAs as poor practices and therefore must be avoided—69% and 51% of the *external* and *internal* developers, respectively. As further evidence of LAs’ validity, open source developers that were made aware of LAs reacted to the issue by making code changes in 10% of the cases. Finally, in order to facilitate the use of LAs in practice, we identified a sub-set of LAs which were universally

Venera Arnaoudova
Soccer Lab., DGIGL, Polytechnique Montréal
2900, boulevard Édouard-Montpetit
2700, chemin de la Tour
Montréal, (Québec) Canada, H3T 1J4
Tel.: +1-514-967-9434
Fax: +1-514-340-5139
E-mail: venera.arnaoudova@polymtl.ca

Massimiliano Di Penta
Department of Engineering, University of Sannio, Benevento, Italy
E-mail: dipenta@unisannio.it

Giuliano Antoniol
Soccer Lab., DGIGL, Polytechnique Montréal, Canada
E-mail: antoniol@ieee.org

¹ This work is an extension of our previous paper [Arnaoudova *et al.*, 2013].

agreed upon as being problematic; those which had a clear dissonance between code behavior and lexicon.

Keywords Source code identifiers · Linguistic antipatterns · Empirical study · Developers’ perception

1 Introduction

There are many recognized bad practices in software development, known as code smells and antipatterns [Brown et al., 1998a; Fowler, 1999]. They concern poor design or implementation solutions, as for example the *Blob*, also known as *God class*, which is a large and complex class centralizing the behavior of a part of the system and using other classes simply as data holders. Previous studies indicate that APs may affect software comprehensibility [Abbes et al., 2011] and possibly increase change and fault-proneness [Khomh et al., 2009, 2012]. From a recent study by Yamashita and Moonen [2013] it is also known that the majority of developers are concerned about code smells.

Most often, documented bad practices deal with the design of a system or its implementation—e.g., code structure. However, there are other factors that can affect software comprehensibility and maintainability, and source code lexicon is surely one of them.

In his theory about program understanding, Brooks [1983] considers identifiers and comments as part of the internal indicators for the meaning of a program. Brooks presents the process of program understanding as a top-down hypothesis-driven approach, in which an initial and vague hypothesis is formulated—based on the programmer’s knowledge about the program domain or other related domains—and incrementally refined into more specific hypotheses based on the information extracted from the program lexicon. While trying to refine or verify the hypothesis, sometimes developers inspect the code in detail, e.g., check the comments against the code. Brooks warns that it may happen that comments and code are contradictory, and that the decision of which indicator to trust (i.e., comment or code) primarily depends on the overall support of the hypothesis being tested rather than the type of the indicator itself. This implies that when a contradiction between code and comments occur, different developers may end up trusting different indicators, and thus have different interpretations of a program.

The role played by identifiers and comments in source code understandability has been also empirically investigated by several other researchers showing that commented programs and programs containing full word identifiers are easier to understand [Chaudhary and Sahasrabudde, 1980; Lawrie et al., 2006, 2007; Shneiderman and Mayer, 1975; Takang et al., 1996; Woodfield et al., 1981].

For the reasons emerged from the above studies, researchers have developed approaches to assess the quality of source code lexicon [Caprile and Tonella, 2000; Lawrie et al., 2007; Merlo et al., 2003] and have provided a set of guidelines to produce high-quality identifiers [Deissenbock and Pizka, 2005]. Also, in a previous work [Arnaoudova et al., 2013], we have formulated the notion of source code Linguistic Antipatterns (LAs), i.e., *recurring poor practices in the naming, documentation, and choice of identifiers in the implementation of an entity* and we defined a catalog of 17 types of LAs related to inconsistencies. The notion of

LAs builds on those previous works. In particular, we concur with *Brooks* [1983] that inconsistencies may lead to different program interpretations by different programmers and some of these interpretation may be wrong, thus impairing program understanding. An incorrect initial interpretation may impact the way developers complete their tasks as their final solution will possibly be biased by this initial interpretation. This cognitive phenomenon is known as *anchoring* and the difficulty to move away (or to adjust) from the initial interpretation as the *adjustment bias*. For example, *Parsons and Saunders* [2004] provide evidence of the existence of the phenomenon in the context of reusing software code/design. Our conjecture is that defining a catalog of LAs will increase developer awareness of such poor practices and thus contributes to the improvement of the lexicon and program comprehension.

An example of an LA, which we have named *Attribute signature and comment are opposite*, occurs in class `EncodeURLTransformer` of the *Cocoon*² project. The class contains an attribute named `INCLUDE_NAME_DEFAULT` whose comment documents the opposite, i.e., a “*Configuration default exclude pattern*”. Whether the pattern is included or excluded is therefore unclear from the comment and name. Another example of LA called “*Get method does not return*” occurs in class `Compiler` of the *Eclipse*³ project where method `getMethodBodies` is declared. Counter to what one would expect, the method does not either return a value or clearly indicate which of the parameters will hold the result.

To understand whether such LAs would be relevant for software developers, two general questions arise:

- Do developers perceive LAs as indeed poor practices?
- If this is the case, would developers take any action and remove LAs?

Indeed, although tools may detect instances of (different kinds of) bad practices, they may or may not turn out to be actual problems for developers. For example, by studying the history of projects *Raïu et al.* [2004] showed that some instances of antipatterns, e.g., *God classes* being persistent and stable during their life, are considered harmless.

This paper aims at empirically answering the questions stated above, by conducting two different studies. In *Study I* we showed to 30 developers an extensive set of code snippets from three open-source projects, some of which containing LAs, while others not. Participants were *external* developers, i.e., people that have not developed the code under investigation, unaware of the notion of LAs. The rationale here is to evaluate how relevant are the inconsistencies, by involving people having no bias—neither with respect to our definition of LAs, nor with respect to the code being analyzed. In *Study II* we involved 14 *internal* developers from 8 projects (7 open-source and 1 commercial), with the aim of understanding how they perceive LAs in systems they know, whether they would remove them, and how (if this is the case). Here, we first introduce to developers the definition of the specific LA under scrutiny, after which they provide their perception about examples of LAs detected in their project.

Overall, results indicate that *external* and *internal* developers perceive LAs as poor practices and therefore should be avoided—69% and 51% of participants

² <http://cocoon.apache.org>

³ <http://www.eclipse.org>

in *Study I* and *Study II*, respectively. Interestingly, developers felt more strongly about certain LAs. Thus, an additional outcome of these studies was a subset of LAs that are considered to be the most problematic. In particular, we identify a subset of LAs i) that are perceived as poor practices by at least 75% of the *external* developers, ii) that are perceived as poor practices by all *internal* developers, or iii) for which *internal* developers took an action to remove it. In fact, 10% (5 out of 47) of the LAs shown to internal developers during the study have been removed in the corresponding projects after we pointed them out. There are three LAs that both external and internal developers find particularly unacceptable. Those are LAs concerning the state of an entity (i.e., attributes) and they belong to the “says more than what it does” (B) and “contains the opposite” (F) categories—i.e., *Not answered question* (B.4), *Attribute name and type are opposite* (F.1), and *Attribute signature and comment are opposite* (F.2). External developers found particularly unacceptable (i.e., more than 80% of them perceived as poor or very poor) the LAs with a clear dissonance between the code behavior and its lexicon—i.e., *“Get” method does not return* (B.3), *Not answered question* (B.4), *Method signature and comment are opposite* (C.2), and *Attribute signature and comment are opposite* (F.2). The extremely high level of agreement on these LAs motivates the need for a tool pointing out these issues to developers while writing the source code.

Paper structure. Section 2 provides an overview of the LA definitions [Arnaoudova et al., 2013] and tooling. Sections 3 describes the definition, design, and planning of the studies. In Section 4 we detail the catalog of LAs while reporting and discussing the results of the two studies. After a discussion of related work in Section 6, Section 7 concludes the paper and outlines directions for future work.

2 Linguistic Antipatterns (LAs)

Software antipatterns—as they are known so far—are opposite to design patterns [Gamma et al., 1995], 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 [Brown et al., 1998b]. 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 [Arnaoudova et al., 2013] 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 mislead developers—they 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 easy to understand code. In other words, our hypothesis is that the quality of the lexicon depends not only on the quality of individual identifiers but also on the consistency among identifiers from different sources (name, implementation, and documentation). Thus, identifying practices that result in inconsistent lexicon, and grouping them into a catalog, will increase developer awareness and thus contribute to the improvement of the lexicon.

Table 1 LAs catalog: Definitions and examples.

A.1	<i>“Get” - more than an accessor</i>	A getter that performs actions other than returning the corresponding attribute without documenting it. Example: method <code>getImageData</code> which, no matter the attribute value, every time returns a new object (see Fig. 1).
A.2	<i>“Is” returns more than a Boolean</i>	The name of a method is a predicate suggesting a true/false value in return. However the return type is not Boolean but rather a more complex type thus allowing a wider range of values without documenting them. Example: <code>isValid</code> with return type <code>int</code> (see Fig. 6).
A.3	<i>“Set” method returns</i>	A set method having a return type different than <code>void</code> and not documenting the return type/values with an appropriate comment (see Fig. 7).
A.4	<i>Expecting but not getting a single instance</i>	The name of a method indicates that a single object is returned but the return type is a collection. Example: method <code>getExpansion</code> returning <code>List</code> (see Fig. 9).
B.1	<i>Not implemented condition</i>	The comments of a method suggest a conditional behavior that is not implemented in the code. When the implementation is default this should be documented (see Fig. 10).
B.2	<i>Validation method does not confirm</i>	A validation method (e.g., name starting with “validate”, “check”, “ensure”) does not confirm the validation, i.e., the method neither provides a return value informing whether the validation was successful, nor documents how to proceed to understand (see Fig. 11).
B.3	<i>“Get” method does not return</i>	The name suggests that the method returns something (e.g., name starts with “get” or “return”) but the return type is <code>void</code> . The documentation should explain where the resulting data is stored and how to obtain it (see Fig. 12).
B.4	<i>Not answered question</i>	The name of a method is in the form of predicate whereas the return type is not Boolean. Example: method <code>isValid</code> with return type <code>void</code> (see Fig. 13).
B.5	<i>Transform method does not return</i>	The name of a method suggests the transformation of an object but there is no return value and it is not clear from the documentation where the result is stored. Example: method <code>javaToNative</code> with return type <code>void</code> (see Fig. 14).
B.6	<i>Expecting but not getting a collection</i>	The name of a method suggests that a collection should be returned but a single object or nothing is returned. Example: method <code>getStats</code> with return type Boolean (see Fig. 15).
C.1	<i>Method name and return type are opposite</i>	The intent of the method suggested by its name is in contradiction with what it returns. Example: method <code>disable</code> with return type <code>ControlEnableState</code> . The inconsistency comes from “disable” and “enable” having opposite meanings (see Fig. 16).
C.2	<i>Method signature and comment are opposite</i>	The documentation of a method is in contradiction with its declaration. Example: method <code>isNavigateForwardEnabled</code> is in contradiction with its comment documenting “a back navigation”, as “forward” and “back” are antonyms (see Fig. 17).
D.1	<i>Says one but contains many</i>	The name of an attribute suggests a single instance, while its type suggests that the attribute stores a collection of objects. Example: attribute <code>target</code> of type <code>Vector</code> . It is unclear whether a change affects one or multiple instances in the collection (see Fig. 18).
D.2	<i>Name suggests Boolean but type does not</i>	The name of an attribute suggests that its value is true or false, but its declaring type is not Boolean. Example: attribute <code>isReached</code> of type <code>int[]</code> where the declared type and values are not documented (see Fig. 19).
E.1	<i>Says many but contains one</i>	The name of an attribute suggests multiple instances, but its type suggests a single one. Example: attribute <code>stats</code> of type Boolean. Documenting such inconsistencies avoids additional comprehension effort to understand the purpose of the attribute (see Fig. 20).
F.1	<i>Attribute name and type are opposite</i>	The name of an attribute is in contradiction with its type as they contain antonyms. Example: attribute <code>start</code> of type <code>MAssociationEnd</code> . The use of antonyms can induce wrong assumptions (see Fig. 21).
F.2	<i>Attribute signature and comment are opposite</i>	The declaration of an attribute is in contradiction with its documentation. Example: attribute <code>INCLUDE_NAME_DEFAULT</code> whose comment documents an “exclude pattern”. Whether the pattern is included or excluded is thus unclear (see Fig. 22).

We defined LAs and group them into categories based on a close inspection of source code examples. We started by analyzing source code from three open-source Java projects—namely ArgoUML, Cocoon, and Eclipse.

Initially, we randomly sampled a hundred files and analyzed the source code looking for examples of inconsistencies of lexicon among different sources of identifiers (i.e., identifiers from the name, documentation, and implementation of an entity). For each file, we analyzed the declared entities (methods and attributes) by asking ourselves questions such as “Is the name of the method consistent with its return type?”, “Is the attribute comment consistent with its name?”. Two of the authors of this paper were involved in the process. The set of inconsistencies examples that we found were then organized into an initial set of LAs. We iterated several times over the sampling and coding process and refine the questions based on the newly discovered examples. For example, “What is an inconsistent name for a boolean return type?”, “Is `void` a consistent type for method `isValid`?”, “What other types are inconsistent with method `isValid`?”, etc.

As the goal is to capture as many different lexicon inconsistencies as possible, the sampling was guided by the theory (theoretical sampling *Strauss* [1987]), and thus cannot be considered to be representative of the entire population of source code entities. We stopped iterating over the sampling and coding process when new examples of inconsistencies did not anymore modify the defined LAs and their categories. Also, it is important to point out that during our analyses we did not follow a thorough grounded-theory approach [*Glaser*, 1992; *Strauss*, 1987]—i.e., we did not measure the inter-agreement at each iteration—as the process was meant to identify possible inconsistencies for which we would gather developers’ perceptions. Thus, the agreement between the authors of this paper was a guidance rather than a requirement. Nevertheless, similarly to grounded-theory, we performed refinements to our initial categories.

Over the iterations LAs were refined, compared, and grouped into categories. Categories were modified according to the new examples of inconsistencies, i.e., some categories were combined, refined, or split to account for the newly defined LAs. For instance, we ended up with a preliminary list of 23 types LAs for which we abstracted the type of inconsistency they concern. At a very high level, all but 6 of those 23 types LAs concerned inconsistencies where i) a single entity (method or attribute) is involved, and ii) the behavior of the entity is inconsistent with what its name or related documentation (i.e., comment) suggests, as it provides more, says more, or provides the opposite. Thus, we discarded from this catalog those LAs that did not obey the above two rules. An example of a discarded LAs from the preliminary list is the practice consisting in the use of synonyms in the entity signature—e.g., method *completeResults(..., boolean finished)*, where the term *complete* in the method name is synonym of *finished* (parameter). Similarly, we discarded the following five cases of comment inconsistencies:

- Not documented or counter-intuitive design decision, e.g., using inheritance instead of delegation.
- Parameter name in the comment is out of date.
- Misplaced documentation, e.g., entity documentation exists, but it is placed in a parent class.
- The entity comment is inconsistent across the hierarchy.

- Not documented design pattern. Previous work has proposed approaches to document design patterns [Torchiano, 2002], and has shown that design patterns’ documentation helps developers to complete maintenance tasks faster and with fewer errors [Prechelt et al., 2002].

The fact that we discarded the above practices from the current catalog does not mean that we consider them any less poor. On the contrary, we believe that further investigation in different projects may result in discovering other related poor practices that can be abstracted into new categories of inconsistencies, thus extending the current catalog.

After pruning out the six types described above, the analysis process resulted in 17 types of LA, grouped into six categories, three regarding behavior—i.e., methods—and three regarding state—i.e., attributes. For methods, LAs are categorized into methods that (A) “do more than they say”, (B) “say more than they do”, and (C) “do the opposite than they say”. Similarly, the categories for attributes are (D) “the entity contains more than what it says”, (E) “the name says more than the entity contains”, and (F) “the name says the opposite than the entity contains”. Table 1 provides a summary of the defined LAs. We further detail the LAs during the qualitative analysis of the results (Section 4.3).

We implemented the LAs detection algorithms in an offline tool, named LAPD (Linguistic Anti-Pattern Detector), for Java source code [Arnaoudova et al., 2013]; for the purpose of this work, we extended the initial LAPD to analyze C++. LAPD analyzes signatures, leading comments, and implementation of program entities (methods and attributes). It relies on the Stanford natural language parser [Toutanova and Manning, 2000] to identify the Part-of-Speech of the terms constituting the identifiers and comments and to establish relations between those terms. Thus, given the identifier `notVisible`, we are able to identify that ‘visible’ is an adjective and that it holds a negation relation with the term ‘not’.

Finally, to identify semantic relations between terms LAPD uses the WordNet ontology [Miller, 1995]. Thus, we are able to identify that ‘include’ and ‘exclude’ are antonyms.

Consider for example the code shown in Fig. 1. To check whether it contains an LA of type “Get” - more than an accessor (A.1) LAPD first analyses the method name. As it follows the naming conventions for accessors—i.e., starts with ‘get’—LAPD proceeds and searches for an attribute named `imageData` of type `ImageData` defined in class `CompositeImageDescriptor`. The existence of the attribute indicates that the implementation of `getImageData` would be expected to satisfy the expectations from an accessor, i.e., return the value of the corresponding attribute. Thus, LAPD analyses the body of `getImageData` and reports the method as an example of “Get” - more than an accessor (A.1) as it contains a number of additional statements before returning the value of `imageData`. Indeed, one can note that the value of the attribute is always overridden (line 69) which is not expected from an accessor except if the value is `null`—as for example the Proxy and Singleton design patterns). Further details regarding the detection algorithms of LAs can be found in Appendix A.

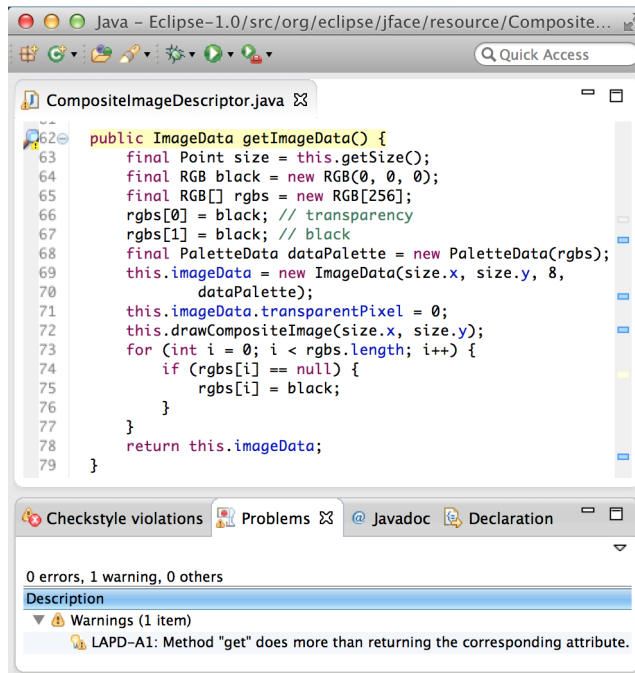


Fig. 1 LAPD Checkstyle plugin: “Get” - more than an accessor (A.1).

For Java source code, we also made available an online version of LAPD⁴ integrated into Eclipse as part of the Eclipse Checkstyle Plugin⁵. Checkstyle⁶ is a tool helping developers to adhere to coding standards, which are expressed in terms of rules (checks), by reporting violations of those standards. Users may choose among predefined standards, e.g., the *Sun* coding conventions⁷, or define their own. Fig. 1 shows a snapshot of a code example and an LA, of type “Get” - more than an accessor (A.1), reported by the LAPD CHECKSTYLE PLUGIN⁴ detected in the example. After analyzing the entity containing the reported LA, the user may decide to resolve the inconsistency or disable the warning report for the particular entity.

3 Experimental design

Before studying the perception of developers, it is important to study the prevalence of LAs. Section 3.1 provides details about our preliminary study on the prevalence of LAs. Next, we report the definition, design, and planning of the two studies we have conducted with external (Section 3.2) and internal (Section 3.3) developers. To report the studies we followed the general guidelines suggested by Wohlin et al. [2000], Kitchenham et al. [2002], and Jedlitschka and Pfahl [2005].

⁴ <http://www.veneraarnaoudova.ca/tools>

⁵ <http://eclipse-cs.sourceforge.net/>

⁶ <http://checkstyle.sourceforge.net/>

⁷ <http://www.oracle.com/technetwork/java/codeconv-138413.html>

Table 2 Preliminary study - Objects characteristics.

Project	Size (LOC)	Language
<i>Apache Maven</i> ⁹ 3.0.5	71 K	Java
<i>Apache OpenMeetings</i> ¹⁰ 2.1.0	52 K	Java
<i>GanttProject</i> ¹¹	57 K	Java
<i>boost</i> ¹² 1.53.0	1.9 M	C++
<i>BWAPI</i> ¹³	118 K	C++
<i>CommitMonitor</i> ¹⁴ 1.8.7.831	148 K	C++
<i>OpenCV</i> ¹⁵	544 K	Java, C++

3.1 Prevalence of LAs

The *goal* of our preliminary 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. Specifically, the preliminary study aims at answering the following research question:

RQ0: *How prevalent are LAs?* We investigate how relevant is the phenomenon of LAs in the studied projects.

Experiment design: For each LA we report the occurrences in the studied projects. We also report the percentage of the programming entities in which an LA occurs with respect to the population for which the LA been defined. Finally, we report the relevance of each LA with respect to the total entity population of its kind. For example, for “*Get*” - *more than an accessor* (A.1) we report the the number of occurrences, the percentage of the occurrences with respect to the number of accessors, and the percentage of the occurrences with respect to all methods.

Objects: Using convenience sampling [Shull et al., 2007], we select seven open-source Java and C++ projects Table 2 shows the projects’ characteristics⁸. We have chosen projects from various application domains, with different size, different programming language, and different number of developers.

Data collection: For this study, we simply downloaded the source code archives of the considered system releases (Table 2), and analyzed them using the LAPD tool with the aim of identifying LAs.

Next, we report the definition, design, and planning of the two studies we have conducted with external (Section 3.2) and internal (Section 3.3) developers. Both studies were designed as online questionnaires. A replication package is available¹⁶. It contains (i) all the material used in the studies, i.e., instructions, ques-

⁸ For projects where we did not provide a version, we used version control (accessed on 31/05/2013).

¹⁶ <http://ser.soccerlab.polymtl.ca/ser-repos/public/tr-data/lapd-rep-pckg.zip>

tionnaires, and LA examples, and (ii) working data sets containing anonymized results for both studies. We do not disclose information about participants' ability and experience.

3.2 Study I (SI)—External Developers

The *goal* of the study is to collect opinions about code snippets containing LAs from the *perspective* of external developers, i.e., people new to the code containing LAs—with the *purpose* of gaining insights about developers' perception of LAs. The feedback of external developers will help us to understand how LAs are perceived by developers who are new to the particular code, as it is often the case when developers join a new team or maintain a large system they are not entirely familiar with. Specifically, the study aims at answering the following research questions:

RQ1: *How do external developers perceive code containing LAs?* We investigate whether developers actually recognize the problem and in such case how important they believe the problem is.

RQ2: *Is the perception of LAs impacted by confounding factors?* We investigate whether results of **RQ1** depend on participant's i) main programming language (for instance Java versus C++, as the LAs were originally defined for Java), ii) occupation (i.e., professionals or students), and iii) years of programming experience.

In the following, we report details of how the study has been planned and conducted.

Experiment design: The study was designed as an online questionnaire estimated to take about one hour for an average of two and a half minutes per code snippet. However, participants were free to take all the necessary time to complete the questionnaire. The use of an online questionnaire was preferred over in person interview, as it is more convenient for the participants. Participants were free to decide when to fill the questionnaire and in how many steps to complete it—i.e., participants may decide to complete the questionnaire in a single session or to stop in between questions and to resume later. To avoid biasing the participants, we also consider as part of the questionnaire 8 code snippets that do not contain any LA. Thus, we ask participants to analyze 25 code snippets (17 being examples of LAs, and 8 not containing any LA), and to evaluate the quality of each example comparing naming, documentation, and implementation. Ideally, we would have preferred to evaluate an equal number of code snippets with and without LAs. This, however, would have increased the required time with more than 20 minutes and increased the chances that participants do not complete the survey. Therefore, we decided to decrease the number of code snippets without LAs by half (compared to the number of code snippets with LAs).

We (the authors) selected examples covering the set of LAs from the analyzed projects that in our opinion are representative of the studied LAs. In particular, we used the examples from our previous work as the study was performed before the proceedings were publicly available.

For each code snippet, we formulated a specific question, trying to avoid any researcher bias on whether the practice is good or poor. Thus, for example, when

showing the code snippet of method `getImageData` (used in Fig. 1) corresponding to the example of “*Get*” - *more than an accessor*; we asked participants to provide their opinion on the practice consisting of *using the word “get” in the name of the method with respect to its implementation*. Note that if the question does not indicate what aspect of the snippet the participants are expected to evaluate, there is a high risk that the participants evaluate an unrelated aspect—e.g., performance or memory related. However, specific questions are subject to the hypothesis guessing bias thus participants may evaluate as poor practices all code snippets as they may guess that this is what is expected. This is why inserting code snippets that do not contain LAs is a crucial part of the design. To compare the scores given by developers to code snippets that contain LAs and those that do not, we perform a Mann-Whitney test.

To minimize the order/response bias, we created ten versions of the questionnaire where the code snippets appear in a random order. Participants were randomly assigned to a questionnaire. To achieve a design as balanced as possible, i.e., equal number of participants for each questionnaire, we invited participants through multiple iterations. That is, we sent an initial set of invitations to an equal number of participants. After a couple of days, we sent a second set of invitations assigning such additional participants to the questionnaire instances that received the lowest number of responses.

Objects: For the purpose of the study, we choose to evaluate LA instances detected and manually validated in our previous work [Arnaoudova et al., 2013]. Such LAs have been detected in 3 Java software projects, namely *ArgoUML*¹⁷ 0.10.1 (82 KLOC) and 0.34 (195 KLOC), *Cocoon*¹⁸ 2.2.0 (60 KLOC), and *Eclipse*¹⁹ 1.0 (475 KLOC).

Participants: Ideally, a target population—i.e., the individuals to whom the survey applies—should be defined as a finite list of all its members. Next, a valid sample is a representative sample of the target population [Shull et al., 2007]. When the target population is difficult to define, non-probabilistic sampling is used to identify the sample. In this study, the target population being all software developers, it is impossible to define such list. We selected participants using convenience sampling. We invited by e-mail 311 developers from open-source and industrial projects, graduate students and researchers from the authors’ institutions as well as from other institutions. 31 developers completed the study and after a screening procedure (see Section 3.2), 30 participants remained—11 professionals, and 19 graduate students, resulting in a response rate close to 10%—as expected [Groves et al., 2009]. Participants were volunteers and they did not receive any reward for the participation to the study. We explicitly told them that anonymity of results was preserved, and so we did.

Table 3 provides information on participants’ programming experience and Fig. 2 shows their native language and the country they live in.

Study procedure: We did not introduce participants to the notion of LAs before the study. Instead, we informed them that the task consists of providing their opinion of code snippets.

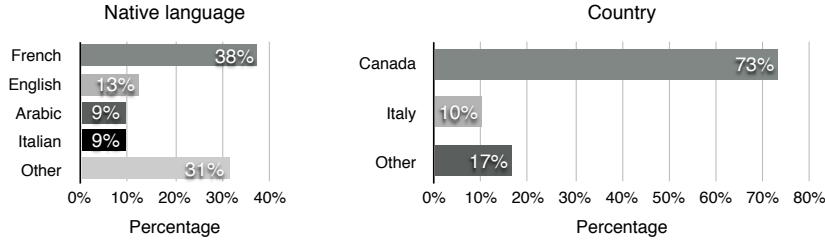
¹⁷ <http://argouml.tigris.org>

¹⁸ <http://cocoon.apache.org>

¹⁹ <http://www.eclipse.org>

Table 3 Study I—programming experience of the participants.

	# of participants	Programming experience (years)	
		< 5	≥ 5
Graduate students	19	9	10
Professionals	11	1	10
Overall	30	10	20

**Fig. 2** Study I—native language and country of the participants.

For each code snippet—containing LAs or not—we asked participants the five questions reported in Table 4. With **SI-q1** participants judge the quality of the practice on a 5-point Likert scale [Oppenheim, 1992], ranging between ‘Very poor’ and ‘Very good’. The purpose of **SI-q2** is to ensure that the participants provide their judgement for the practice targeted by the question. For both **SI-q4** and **SI-q5**, we provide predefined options, to decrease the effort and ease the analysis, however we left space in the form to provide a customized answer. In addition, for each code snippet, we also allow participants to share any additional comment they would make. At any point, participants are free to decide not to answer a question by selecting the option ‘No opinion’.

Data Collection: We collected 31 completed questionnaires. Before proceeding with the analysis, we applied the following screening procedure: For each LA we remove subjects who chose ‘No opinion’ as answer to **SI-q1**.

The collected answers being in nominal and ordinal scales, standard outlier removal techniques do not apply here. Thus, we first sought for inconsistent answers between questions **SI-q1** and **SI-q3**, i.e., between the quality of the code snippet and whether an action should be undertaken. Although one may judge a code snippet as ‘Poor’ but believes that no action should be undertaken, we fear that participants providing high number of such combinations may have misunderstood the questions. We intentionally sought for participants providing high number of such combinations (> 75%), resulting in removing one participant.

Then, we individually analyze the justification, i.e., answers of **SI-q2**, and we remove the answer of a participant for an LA if it is clear that the participant judge an aspect different from the one targeted by the LA. For example, when a participants are asked to give their opinion on the use of conditional sentence in comments and no conditional statement in method implementation, participant providing the following justification is removed for the particular LA: *“the method name is well chosen and is well commented too”*. Thus, the number of obtained

Table 4 Study I - Questionnaire.

Question	Possible answers
SI-q1: You judge this practice as:	(Single choice) Very poor Poor Neither poor nor good Good Very good No opinion
SI-q2: Please justify	(Free-form)
SI-q3: Would you undertake an action with respect to the practice?	(Single choice) Change Keep it ‘as is’ No opinion
SI-q4: Illustrate the kind of action you would undertake (when this is the case).	(Multiple choice) Comments (add/remove/modify) Renaming Implementation (add/remove/modify) Other ---
SI-q5: Explain the reason why you would not undertake any action (when this is the case).	(Multiple choice) It is a common practice Naming and functionality are consistent Comments and naming are consistent Comments and functionality are consistent Other ---

answers for each kind of LA varies between 25 and 30, as it can be noticed from Fig. 5.

3.3 Study II (SII)—Internal Developers

The *goal* of this study is to investigate the perception of LAs from the *perspective* of *internal* developers, i.e., those contributing to the project in which LAs occur. Internal developers will provide us not only with their opinion about LAs but also with insights on the typical actions they are willing to undertake, to correct the existing inconsistencies and possibly help us to understand what causes LAs to occur. The *context* consists of examples of code, selected from projects to which the surveyed developers contribute. To extend the external validity of the results, for this study, we considered projects written in two different programming languages, i.e., Java and C++. The study aims at answering the following research questions:

RQ3: *How do internal developers perceive LAs?* This research question is similar to RQ1 of *Study I*, however here we are interested in the perception of developers familiar with the code containing LAs, i.e., of people who contributed to it.

RQ4: *What are the typical actions to resolve LAs?* Other than the opinion on the practices described by LAs, we investigate whether developers are willing to undertake actions to correct the suggested inconsistencies.

RQ5: *What causes LAs to occur?* We are interested to understand under what circumstances LAs appear to better cope with them.

Experiment design: The study was designed as an online questionnaire. The number of LAs was selected so that the questionnaire requires approximately 15 minutes to be completed, and therefore ensures a high response rate from internal developers. As in *Study I*, the time was simply indicative, i.e., participants are free to take all the necessary time to complete the questionnaire. As LAs were related to methods having different size and complexity, the questionnaires contained between 5 and 6 examples, i.e., not always the same number. Thus, each participant evaluates only a subset of the LAs. We, the authors, selected examples of LAs from the analyzed projects that in our opinion are representative of the studied LAs. We selected the examples in a way to have higher diversity, i.e., so that the study includes examples of all 17 types of LAs.

Objects: To select the projects for this study we also used a convenience sampling. We consider LAs extracted from 8 software projects, specifically 1 industrial, closed-source project, namely *MagicPlan*²⁰, and 7 open-source projects—i.e., the projects used in the preliminary study (see Table 2). The projects have different size and belong to different domain, ranging from utilities for developers/project managers (e.g., *Apache OpenMeetings*, *GanttProject*, *commitMonitor*, *Apache Maven*) to APIs (*Boost*, *BWAPI*, and *OpenCV*) or mobile applications (*MagicPlan*). We chose more projects than in *Study I*, in order to obtain a larger external validity from developers belonging to different projects (including a commercial one), and in order to consider both Java and C++ code.

Participants: The study involved 14 developers from the projects mentioned above. As for the distribution across projects, one developer per project participated in the study, except for *Boost*, for which 3 developers participated, and for *BWAPI*, for which 4 developers participated. Such 14 developers are the respondents from an initial set of 50 ones we invited to participate, resulting in a response rate of 28%. Invited participants were committers whose e-mails were available in the version control repository of the project. Also in this case, participants were volunteers and did not receive any reward. Similarly to the previous study, anonymity of results was preserved.

Study procedure: We showed to participants examples of LAs detected in the system they contribute to. For each example, we first provided participants with the definition of the corresponding LA, and then we asked them to provide an opinion about the general practice—i.e., question **SII-q0** “How do you consider the practice described by the above Linguistic Antipattern?”—using, again, a 5-point Likert scale. Then, we asked participants to provide indications about the specific instance of LA by asking the questions shown in Table 5.

²⁰ <http://www.sensopia.com/english/index.html>

Table 5 Study II - Questionnaire.

Question	Possible answers
SII-q1: How familiar are you with this code?	(Single choice) I wrote it I didn't write it but I came across this code Don't remember seeing it before Other ---
SII-q2: Why the inconsistency occurred, i.e., what are the causes?	(Multiple choice) Evolution (it was consistent initially) Didn't give it enough thought initially Copy/paste and forgot to change Reuse without changing since it is working Other ---
SII-q3: Equivalent to SI-q3	Equivalent to SI-q3
SII-q4: Equivalent to SI-q4	(Free-form)
SII-q5: Equivalent to SI-q5	Equivalent to SI-q5

Data Collection: We collected responses of 14 developers regarding 47 unique examples of all types of LAs except C.2²¹. The collected answers represent 72 data points, where each data point is a unique combination of a particular example (instance) of an LA and a developer who evaluated it.

4 Developers' Perception of LAs

In this section we present the results of our studies. First, in Section 4.1 we report the results of our preliminary study on the prevalence of LAs. Next, we present the results of the two studies on developers' perceptions of LAs providing both quantitative (Section 4.2) and qualitative (Section 4.3) analyses.

4.1 Prevalence of LAs

RQ0: How prevalent are LAs?

Table 6 shows the number of detected instances of LAs per project and per kind of LA. Based on previous evaluation on a subset of those systems, LAPD has an average precision of 72% (95% confidence level and a confidence interval of $\pm 10\%$). As the goal of this work is to evaluate developers' perception of LAs we did not re-evaluate the precision but rather manually validated a subset of the detected examples to assure that they are indeed representative of LAs.

Table 7 shows how relevant is the phenomenon in the studied projects. For each LA we report its relevance with respect to the population for which it has been defined as well as its relevance with respect to the total entity population of its kind. For example, the first row of Table 7—“*Get*” - *more than an accessor* (A.1)—shows that such complex accessors represent 2.65% of the accessors and

²¹ None of the questionnaires containing examples of type C.2 was answered.

Table 6 LAs : Detected occurrence in the studied projects.

		<i>ArgoUML 0.10.1</i>	<i>ArgoUML 0.34</i>	<i>Cocoon 2.2.0</i>	<i>Eclipse 1.0</i>	<i>Apache Maven 3.0.5</i>	<i>Apache OpenMeetings 2.1.0</i>	<i>GanttProject</i>	<i>boost 1.53.0</i>	<i>BWAPI</i>	<i>CommitMonitor 1.8.7.831</i>	<i>OpenCV</i>	Total
A.1	<i>"Get" - more than an accessor</i>	0	2	1	15	6	2	2	0	0	1	36	65
A.2	<i>"Is" returns more than a Boolean</i>	2	0	4	26	1	0	5	137	2	36	33	246
A.3	<i>"Set" method returns</i>	4	30	6	53	314	29	9	6	73	50	67	641
A.4	<i>Expecting but not getting a single instance</i>	7	3	8	33	40	78	42	16	0	0	5	232
B.1	<i>Not implemented condition</i>	20	28	43	232	2	1	1	1	0	9	3	340
B.2	<i>Validation method does not confirm</i>	1	8	11	235	27	1	0	297	4	18	19	621
B.3	<i>"Get" method does not return</i>	1	3	2	57	17	5	3	0	0	0	0	88
B.4	<i>Not answered question</i>	0	2	0	34	0	0	1	5	0	0	3	45
B.5	<i>Transform method does not return</i>	0	86	15	44	1	4	0	46	11	24	177	408
B.6	<i>Expecting but not getting a collection</i>	8	39	12	135	14	27	19	12	55	3	16	340
C.1	<i>Method name and return type are opposite</i>	0	0	0	6	0	1	2	15	2	0	0	26
C.2	<i>Method signature and comment are opposite</i>	7	20	12	243	7	68	8	55	44	288	105	857
D.1	<i>Says one but contains many</i>	15	92	42	103	42	31	102	1272	219	47	825	2790
D.2	<i>Name suggests Boolean but type does not</i>	14	13	21	138	9	25	11	89	171	151	194	836
E.1	<i>Says many but contains one</i>	45	117	24	116	13	7	6	305	77	388	680	1778
F.1	<i>Attribute name and type are opposite</i>	1	0	0	0	0	0	2	528	0	5	5	541
F.2	<i>Attribute signature and comment are opposite</i>	1	0	3	19	0	1	0	9	3	88	94	218
		126	443	204	1489	493	280	213	2793	661	1108	2262	10072

0.05% of all methods. By looking at the table, the percentage of LAs instances may appear rather low (Min.: 0.02%; 1st Qu.: 0.17%; Median: 0.26%; Mean: 0.61%; 3rd Qu.: 0.65%; Max.: 3.40%). However, in their work on smell detection using change history information *Palomba et al.* [2013] provide statistics about the number of actual classes involved in 5 types of code smells in 8 Java systems; the percentages of affected classes are below 1% for each type of smell, thus somewhat consistent with our findings—although a direct comparison is difficult (due to the different types of entities) the numbers can be taken as a rough indication. Slightly higher are the statistics provided by *Moha et al.* [2010] in which for 10 systems the percentage of affected classes for 4 design smells are as follows: Blob: 2.8%, Functional Decomposition: 1.8%, Spaghetti Code: 5.5%, and Swiss Army Knife: 3.9%.

Table 7 LAs : Relevance of the phenomenon in the studied projects.

		Relevance of the phenomenon	Considered population	Relevance with respect to the entities of the same kind
A.1	<i>“Get” - more than an accessor</i>	2.65% (65/2457)	getters	0.05% (65/129984)
A.2	<i>“Is” returns more than a Boolean</i>	7.44% (246/3307)	methods starting with 'is'	0.19% (246/129984)
A.3	<i>“Set” method returns</i>	10.95% (641/5855)	methods starting with 'set'	0.49% (641/129984)
A.4	<i>Expecting but not getting a single instance</i>	1.72% (232/13527)	methods expecting single instance to be returned	0.18% (232/129984)
B.1	<i>Not implemented condition</i>	6.39% (340/5317)	methods having a documented condition	0.26% (340/129984)
B.2	<i>Validation method does not confirm</i>	69.31% (621/896)	validation method	0.48% (621/129984)
B.3	<i>“Get” method does not return</i>	0.52% (88/17065)	methods whose name suggest that a result will be returned	0.07% (88/129984)
B.4	<i>Not answered question</i>	1.19% (45/3783)	methods whose name suggest Boolean value as a result	0.03% (45/129984)
B.5	<i>Transform method does not return</i>	19.33% (408/2111)	transform method	0.31% (408/129984)
B.6	<i>Expecting but not getting a collection</i>	23.35% (340/1456)	methods whose name suggest that a collection is returned	0.26% (340/129984)
C.1	<i>Method name and return type are opposite</i>	0.02 % (26/129984)	methods	0.02% (26/129984)
C.2	<i>Method signature and comment are opposite</i>	2.53% (857/33910)	documented methods	0.66% (857/129984)
D.1	<i>Says one but contains many</i>	5.98% (2790/46624)	the number of attributes whose name suggests that it contains a single object	3.41% (2790/81886)
D.2	<i>Name suggests Boolean but type does not</i>	64.31% (836/1300)	then number of attributes whose name suggest that it contains a boolean value	1.02% (836/81886)
E.1	<i>Says many but contains one</i>	69.53% (1778/2557)	attributes whose names suggest plural	2.17% (1778/81886)
F.1	<i>Attribute name and type are opposite</i>	0.66% (541/81886)	attributes	0.66% (541/81886)
F.2	<i>Attribute signature and comment are opposite</i>	1.18% (218/18498)	documented attributes	0.27% (218/81886)

Moreover, when we consider only the relevant population, the phenomenon appears to be sufficiently important to justify our interest (Min.: 0.02%; 1st Qu.: 1.19%; Median: 5.98%; Mean: 16.89%; 3rd Qu.: 19.33%; Max.: 69.53%).

In the rest of this section we present the results of both studies providing both quantitative (Section 4.2) and qualitative (Section 4.3) analyses.

4.2 Quantitative analysis

Quantitative analysis pertain all **RQs** from **RQ1** to **RQ5**. In **RQ1** and **RQ2** we report the results from the study with external developers, i.e., *Study I*, while

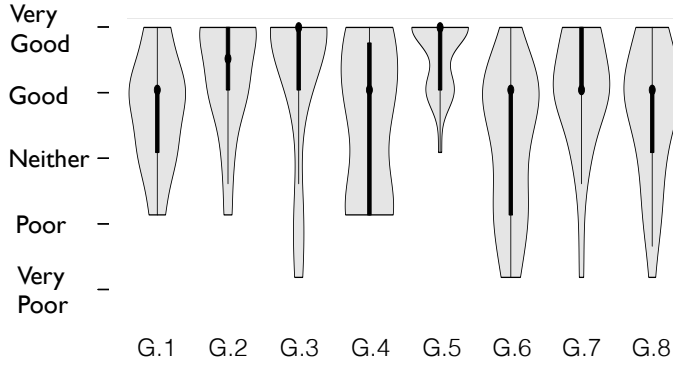


Fig. 3 Violin plots representing how participants perceive examples without LAs.

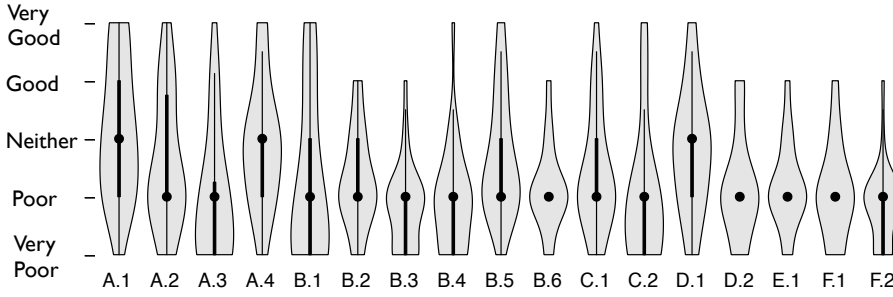


Fig. 4 Violin plots representing how participants perceive LAs.

in **RQ3** to **RQ5** we report the results from the study with internal developers, i.e., *Study II*.

RQ1: How do external developers perceive code containing LAs?

We first analyzed the developers' perception of examples without LAs. Fig. 3 shows violin plots [Hintze and Nelson, 1998] depicting the developers' perception of examples without LAs. Violin plots combine boxplots and kernel density functions, thus providing a better indication of the shape of the distribution. The dot inside a violin plot represents the median; a thick line is drawn between the lower and upper quartiles; a thin line is drawn between the lower and upper tails. As expected, those examples are perceived as having a median 'Good' quality (1st quartile: 'Neither good nor poor', median: 'Good', 3rd quartile: 'Very good').

Fig. 4 shows violin plots depicting the developers' perception of LAs individually for each kind—having a median 'Poor' quality (1st quartile: 'Poor', median: 'Poor', 3rd quartile: 'Neither good nor poor'). Mann-Whitney test indicates that the median score provided for code without LAs is significantly higher than for code with LAs (p -value < 0.0001), with a large ($d = 0.66$) Cliff's delta (d) effect size [Grissom and Kim, 2005]. Overall, if we consider all LAs, 69% of the participants perceive LAs as 'Poor' or 'Very Poor' practices. However, as Fig. 4 shows, the perception distribution varies among different LAs. For instance, boxplots—i.e., the

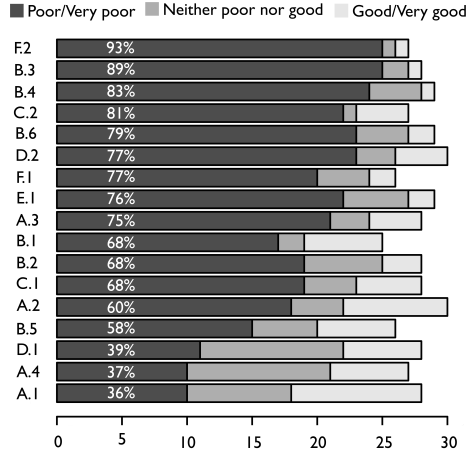


Fig. 5 Percentage of participants perceiving LAs as ‘Poor’ or ‘Very Poor’.

inner lines of violin plots—for A.3 (*“Set” method returns*), B.1 (*Not implemented condition*), B.3 (*“Get” method does not return*), B.4 (*Not answered question*), C.2 (*Method signature and comment are opposite*), and F.2 (*Attribute signature and comment are opposite*) have lower quartile at ‘Very Poor’, median at ‘Poor’, and, for all of them except B.1, higher quartile at ‘Poor’.

We also observe that the perceptions of B.6 (*Expecting but not getting a collection*), D.2 (*Name suggests Boolean but type does not*), E.1 (*Says many but contains one*), and F.1 (*Attribute name and type are opposite*) have little variability and are generally ‘Poor’.

On the contrary, the most controversial LAs are A.1 (*“Get” - more than an accessor*) and A.2 (*“Is” returns more than a Boolean*), with lower and higher quartiles being at ‘Poor’ and ‘Good’ respectively. Other controversial LAs are A.4 (*Expecting but not getting a single instance*), B.2 (*Validation method does not confirm*), B.5 (*Transform method does not return*), C.1 (*Method name and return type are opposite*), and D.1 (*Says one but contains many*), with lower and higher quartiles being at ‘Poor’ and ‘Neither poor nor good’ respectively.

In addition to violin plots, we show proportions of the LA perception by grouping, on the one hand, ‘Poor’ and ‘Very Poor’ judgements, and on the other hand, ‘Good’ and ‘Very Good’ ones. Results are reported in Fig. 5, where we sort LAs based on the proportion of participants that perceive them as ‘Poor’ or ‘Very Poor’. We observe that, for all but three LAs, majority of participants perceive LAs as ‘Poor’ or ‘Very Poor’. The three exceptions are A.1 (*“Get” - more than an accessor*), A.4 (*Expecting but not getting a single instance*), and D.1 (*Says one but contains many*), for all of which the percentage of participants perceiving them as ‘Poor’ or ‘Very Poor’ is 36%, 37%, and 39%, respectively. These are the three LAs having a median perception of ‘Neither poor nor good’ (see Fig. 4).

RQ2: Is the perception of LAs impacted by confounding factors?

We grouped the results of the participants according to their (i) main programming language (Java/C# or C/C++), (ii) occupation (student vs. professional),

and (iii) years of programming experience (< 5 or ≥ 5 years). The grouping concerning the main programming language is motivated by the different way the languages handle Boolean expressions i.e., in C/C++ an expression returning a non-null or non-zero value is evaluated as true, whereas Java and C# do not perform this cast directly. For this reason, our conjecture is that developers who are used to C/C++ would consider acceptable that a method/attribute that should return/contain a Boolean could instead return/contain an integer.

We performed a Mann-Whitney test to compare the median perception of participants in each group. Results regarding the main programming language and the experience of participants indicate no significant difference ($p\text{-value} > 0.05$) with a negligible Cliff’s delta effect size ($d < 0.147$). We obtained consistent results—i.e., no statistically significant differences when analyzing each LA separately—thus, neither the main programming language nor the experience affect the way participants perceive LAs. Only when considering LAs separately, the difference between the rating given by professionals and students is statistically significant for D.2—i.e., *Name suggests Boolean but type does not*, $p\text{-value} = 0.049$, with a medium effect size ($d = 0.40$)—and a marginally significant for E.1—i.e., *Says many but contains one*, $p\text{-value} = 0.053$, with a medium effect size ($d = 0.39$). Thus, we conclude that developers’ perceptions of LAs are not impacted by their main programming language, occupation, or experience.

With the remaining three research questions we investigate the perception of LAs of internal developers—i.e., we report the results of *Study II*.

RQ3: How do internal developers perceive LAs?

Regarding the general opinion of participants (i.e., answers of **SII-q0**), 51% of the times participants perceived LAs as ‘Poor’ or ‘Very poor’. This percentage is lower than the one obtained in *Study I* with external developers, i.e., 69%. In our understanding—and also according to what we observed from developers’ comments (see Section 4.3)—such a decrease in the proportion may sometimes be due to the context in which LAs occur where internal developers perceive LAs as acceptable.

RQ4: What are the typical actions to resolve LAs?

Participants would undertake an action in 56% of the cases, and in 44% of the cases they believe that the code should be left ‘as is’. We discuss the reasons behind these two choices—as reported by the participants—and illustrate them with examples in Section 4.3²².

Overall, the kind of changes that participants are willing to undertake to reduce the effect of LAs fall into one of the following (or a combination of those) categories: renaming, change²³ in comments, and change in implementation. In 42% of the cases, the solution involved renaming, 14% involved a change of comments, and

²² We do not report project names with the examples to avoid disclosing the confidentiality of the provided answers.

²³ A change may be one or more of the following: modification, addition, or removal.

11% a change in the implementation. Resolving an LA may involve changes from the different categories.

10% (5 out of 47) of the LAs shown to internal developers during the study have been removed in the corresponding projects after we pointed them out. The removed examples were instances of A.2 (*“Is” returns more than a Boolean*), A.3 (*“Set” method returns*), B.2 (*Validation method does not confirm*), and B.4 (*Not answered question*). We report the examples in the corresponding LA tables when discussing the perception of internal developers.

Clearly, one must consider that whether or not developers would actually undertake an action depends on other factors such as the potential impact on other projects, the risk of introducing a bug, and the high effort that is required [Arnaoudova et al., 2014]. Sometimes, developers are reluctant to rename programming entities that belong to non-local context (e.g., public methods) or that are bound at runtime (e.g., when classes are loaded by name or methods are bound by name). We believe that some of those factors can be mitigated if LAs are pointed out as developers write source code thus, for example, removing or limiting the impact on other code entities.

RQ5: What causes LAs to occur?

Regarding the possible causes of LAs, we limit our analysis only to cases where the participants wrote the code containing the LAs and cases where they were knowledgeable of that code, e.g., because they were maintaining it. The reported causes and the number of times they occur are as follows (ordered by decreasing order of frequency):

1. *Evolution* (8): The code was initially consistent, but at some point an inconsistency was introduced, hence causing the LA.
2. *Developers’ decision* (7): It is a design choice or simply personal preference.
3. *Not enough thought* (5): Developers did not carefully choose the naming when writing the code.
4. *Reuse* (2): Code was reused from elsewhere without properly adapting the naming.

4.3 Qualitative analysis

For each type of LA, we first briefly summarize its definition and we provide the rationale behind it. Then, we illustrate it using the example we showed to external developers (i.e., in *Study I*)—examples coming from real software projects— followed by possible consequences and solutions. Next, we highlight the perception of external and internal developers. Finally, we report the causes of LA introduction—when reported by the internal developers.

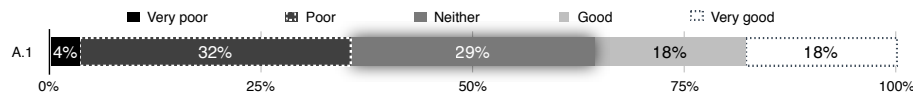
A.1 - “Get” - more than an accessor

A getter that performs actions other than returning the corresponding attribute without documenting it.

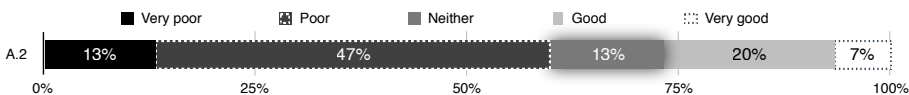
Rationale	In Java, accessor methods, also called getters, provide a way to access class attributes. As such, it is not common that getters perform actions other than returning the corresponding attribute. Any other action should be documented, possibly naming the method differently than <code>getSomething</code> .
Example	Method <code>getImageData</code> which, no matter the attribute value, every time returns a new object (Fig. 1).
Consequences	The usage of such getters would cause an 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.
Example solution	When additional actions in accessor methods are necessary they need to be documented except for common practices—e.g., when implementing a lazy initialization [Gamma et al., 1995]. A possible solution for the example shown in Fig. 1 is to rename the method to <code>createImageData</code> and to comment the unusual behavior: “A new <i>ImageData</i> object is created and assigned to the attribute every time the method is called”.

Developers’ perception

External As shown in the stacked bar chart below, 36% (10 participants), perceived the practice as ‘Poor’ or ‘Very Poor’. 8 participants, i.e., 29%, perceived the practice as ‘Neither poor nor good’, while they suggested renaming and/or refactoring actions. Finally, 10 participants, i.e., 36%, perceived this practice as ‘Good’ or ‘Very good’ “because this is common practice”, and, 3 of those commented that the documentation should specify the additional functionality.



Internal	Developers decided not to refactor the examples of this type. For instance, regarding method <code>getPhases</code> which retrieves a result rather than being a simple accessor, one of the developers commented on the decision not to change it: “perhaps the method could be renamed to <i>findPhasesForLifecycle</i> , but if I remember correctly this class is meant as a data store and then the getter is fine”.
Causes	Developers’ decision.

A.2 - “Is” returns more than a Boolean	
Method name is a predicate, whereas the return type is not Boolean but a more complex type allowing a wider range of values.	
Rationale	When a method name starts with the term “is” one would expect Boolean as return type, thus having two possible values for the predicate, i.e., <code>true</code> and <code>false</code> . Thus, having an “is” method that does not return Boolean, but returns more information is counterintuitive. In such cases, the method should be renamed or, at least, details about the return values should be included in the method comments.
Example	Method <code>isValid</code> with return type <code>int</code> (see Fig. 6).
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 on the maintainers’ side.
Example solution	When the return type cannot be changed to Boolean, we recommend to document the return values. An example of documentation for the method shown in Fig. 6, is “ <i>The method returns −1 for ‘invalid’, 1 for ‘valid’, and 0 for ‘don’t know’</i> ”.
Developers’ perception	
External	18 participants (60%)—i.e., the majority—perceived this practice as ‘Poor’ or ‘Very Poor’ and would have preferred a Boolean return type. 4 participants (13%) perceived this practice as ‘Neither poor nor good’. However, 8 participants (27%) perceived this practice as ‘Good’ or ‘Very Good’. Interestingly, 3 of them explicitly referred to the return values being 0 or 1, and indicated that they are commonly used instead of the Boolean values <code>false</code> and <code>true</code> . However, the particular method returns −1 (which corresponds to “invalid”), 1 (“valid”), or 0 (“don’t know”).
	
Internal	Developers resolved the inconsistency of method <code>isLeft</code> returning <code>float</code> , by removing the method (because the method was replaced by a different one) after the forgotten call to <code>isLeft</code> was replaced with the new method. The developer explained that the method was reused from elsewhere and the name was not adapted after the functionality changed.
Causes	<i>Evolution , reuse.</i>

A.3 - “Set” method returns

A set method having a return type different than void and not documenting the return type/values with an appropriate comment.

Rationale

Modifier methods, also called setters, are methods that allow assigning a value to a class attribute (the attribute being 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 to avoid any misuse. Valid exceptions of this practice are returning the modified attribute after the modification, or returning the object in which the method is defined to allow chaining method calls.

Example

Method `setBreadth` in Fig. 7 shows one such case where the method always creates a new object and returns it.

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.

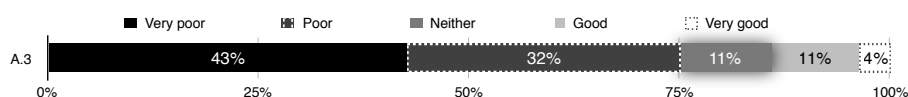
Example solution

The example shown in Fig. 7 can be improved by documenting that “*The method creates a `Dimension` and set its breadth to the value of `source`.*” and by renaming the method to `createDimensionWithBreadth`.

Developers’ perception

External

The majority—21 participants (75%)—perceived this practice as ‘Poor’ or ‘Very Poor’ from which 12 participants (43%) perceived this practice as ‘Very poor’. 3 participants (11%) perceived this practice as ‘Neither poor nor good’ and 4 participants (14%) perceived this practice as ‘Good’ or ‘Very Good’. A participant indicated that in OO programming “*majority of coders will agree that the word ‘set’ is usually used in opposition with ‘get’ so that many coders will suppose this method is setting a value to a member/attribute. This is a very poor practice since this function is not setting anything but instead creating an object*”. The only participant that perceived this practice as ‘Very good’ justified that returning a value from a ‘set’ method can have a benefit as “*in most languages except Java it allows for chaining of method calls*”.

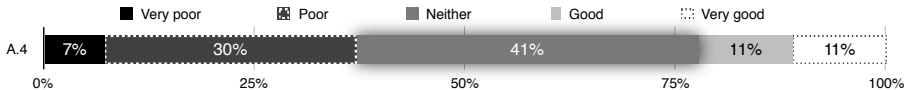


Internal

A developer commented: “*Sometimes it is convenient that a ‘set’ method returns the old or the new value*”. However, two of the LA instances that internal developers resolved after we pointed out the inconsistency were of type A.3. One occurred in method `setConnectionAsSharingClient` returning `Map`; the LA was resolved by improving the (*Javadoc*) documentation, explaining the return type and values. The other instance occurred in method `setAnimationView`, returning `AnimationView`. The changes applied to resolve it impacted 3 files (see Fig. 8). The inconsistency was resolved by: i) improving the *Javadoc* explaining that the old value of the attribute is returned (class `NotificationManager`) ii) renaming the local variable `result` to `oldView` in the child class to reflect that the result contains the old value (class `NotificationManagerImpl`), and iii) renaming the attribute `myAnimationView` to `myOriginalAnimationView` in class `DialogImpl` which contains the result of `setAnimationView`, to reflect that it contains the old value.

Causes

Evolution.

A.4 - Expecting but not getting a single instance	
Method name indicates that a single object is returned but the return type is a collection.	
Rationale	When a method name indicates that a single object is returned, one would expect that a single object is also returned. If, instead, the return type is a collection, the method shall be renamed or appropriate documentation is needed.
Example	Method <code>getExpansion</code> returning <code>List</code> (see Fig. 9)—defined in class <code>DrillFrame</code> —suggests that an object <code>Expansion</code> will be returned whereas a collection is.
Consequences	Although this would unlikely cause faults at run-time, it might cause false expectancies to the developers. When reading <code>getExpansion</code> , one would expect to handle a simple object, whereas it is necessary to deal with multiple objects, which requires different source code to analyze the result, e.g., iterators.
Example solution	A possible solution for the method shown in Fig. 9 would be to rename it to <code>getTreeNodes</code> and rename the attribute accordingly.
Developers' perception	
External	10 participants (37%) perceived this practice as ‘Poor’ or ‘Very poor’. 11 participants (41%) perceived this LA as ‘Neither poor nor good’. 6 of them justified that in the particular case, ‘expansion’ can be considered as ‘list’, hence it does not require plural. The other 5 would undertake a renaming. 6 participants (22%) considered this LA as ‘Good’ or ‘Very good’, and also justified that ‘expansion’ suggests a collection, or that they would understand the code by inferring the presence of a collection from the return type or from the comment.
	
Internal	Developers expressed the need to rename method <code>getMeetingMember</code> returning <code>List<MeetingMemberDTO></code> and <code>getAppointmentByRange</code> returning <code>List<Appointment></code> ; and to comment method <code>getServersOption</code> returning <code>ListOption<WebDavServerDescriptor></code> .
Causes	<i>Evolution.</i>

```

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();
}

```

Fig. 6 “Is” returns more than a Boolean (A.2).

```

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

```

Fig. 7 “Set” method returns (A.3).

<pre> Class NotificationManager 30 + /** 31 + * Sets the animation view of the manager to the given view. 32 + * @return the old view 33 + */ 38 34 AnimationView setAnimationView(AnimationView view); </pre>	<pre> Class NotificationManagerImpl 154 + @Override 154 155 public AnimationView setAnimationView(AnimationView view) { 155 - AnimationView result = myAnimationView; 156 + AnimationView oldView = myAnimationView; 156 157 myAnimationView = view; 157 - return result; 158 + return oldView; 158 159 } </pre>
<pre> Class DialogImpl 119 - private AnimationView myAnimationView; 119 + /** Original animation view, used to set it back when the dialog is closed again */ 120 + private AnimationView myOriginalAnimationView; </pre>	

Fig. 8 Changes applied to resolve an occurrence of A.3—setAnimationView.

```

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

```

Fig. 9 Expecting but not getting a single instance (A.4).

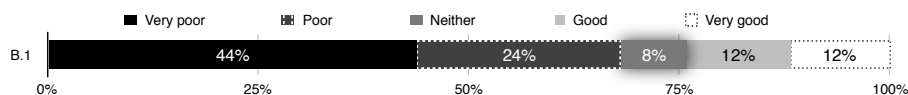
B.1 - Not implemented condition

The method's comments suggest a conditional behavior that is not implemented in the code. When the implementation is default this should be documented.

Rationale	A leading comment summarizes the behaviour of a method at a higher level of abstraction. It allows developers to grasp the intent of the method and the main lines of the implementation without the need to go over all statements in the method's body. Thus, when a condition is expressed in a method's comment one assumes that the condition is implemented.
Example	Fig. 10 shows a method defined in class <code>FileEditionEditorInput</code> that based on the comment <i>"returns an empty array if this object has no children"</i> whereas the implementation 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).
Example solution	The method shown in Fig. 10 could document the default behavior if it is intentional: <i>"This method provides a default behavior by always returning an empty array."</i>

Developers' perception

External	17 participants (68%) perceived this practice as 'Poor' or 'Very Poor'—11 of which (44%) perceived this practice as 'Very poor'. Some of them assumed that the implementation is a placeholder for future code and explained <i>"...that's really dangerous! Such code builds perfectly and sooner or later will be used by someone who will have a very bad surprise about the results"</i> . 2 participants (8%) perceived this practice as 'Neither poor nor good' and 6 participants (24%) perceived this practice as 'Good' or 'Very Good'.
-----------------	--



Internal	The example we pointed out is documented as: <i>"Release the current detector and load new detector from file (if detector_file_name is not 0). Return true on success."</i> , whereas its implementation always returns <i>false</i> . The developer shared that <i>"the code is part of a legacy module and it will be removed with the next major library update"</i> .
-----------------	--

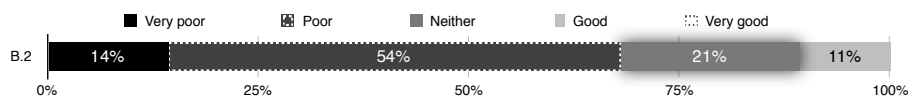
B.2 - Validation method does not confirm

A validation method that neither provides a return value informing whether the validation was successful, nor it documents how to proceed to understand.

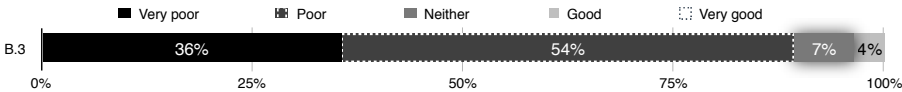
Rationale	A validation method—i.e., a method whose name starts with, for example, “validate”, “check”, or “ensure”—is a method performing a check for validity that is usually required as a precondition for other operations. As such, validation methods are expected to inform the user whether the check is successful or not either by returning true/false or by throwing an exception in case the validation fails.
Example	Fig. 11 shows method <code>checkCollision</code> defined in class <code>UMLComboBoxEntry</code> that neither returns 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.
Example solution	Validation methods must inform the user of the result of the validation by means of return value, exceptions, warnings, or errors. A solution for the example in Fig. 11 would be to return the variable <code>collision</code> . In addition, the actions in case of collision could be extracted in a separate method named <code>resolveCollision</code> .

Developers' perception

External	The majority of the participants, i.e., 19 participants (68%), agreed that this is a poor/very poor practice. The remaining 9 participants were more lenient—3 participants (11%) perceived it as ‘Good’ and 6 participants (21%) as ‘Neither poor nor good’. This is mainly because they trust the validation performed by the method, and do not expect a return value. Indeed, one of them explained that “ <i>Would be better to have a value that would certify the results but not necessary</i> ”.
-----------------	---



Internal	Method <code>validateSnaps</code> with return type <code>void</code> , is an example of B.2, that was renamed to <code>processSnaps</code> after we pointed out the inconsistency. Other examples of this LAs where developers expressed a need for renaming are methods <code>checkVertices</code> and <code>checkCurrentState</code> . The 2 examples where developers decided not to take an action are method <code>validateActivatedProfiles</code> which in case of invalid profile notifies the user with a warning; method <code>checkRecordingFile</code> where the developer commented “ <i>a method that starts with the name “check” has a special validation meaning is new to me.</i> ”
Causes	<i>Evolution, not enough thought.</i>

B.3 - “Get” method does not return	
The name suggests that the method returns something (e.g., name starts with “get” or “return”) but the return type is <code>void</code> . The documentation should explain where the resulting data is stored and how to obtain it.	
Rationale	A method whose name starts with “get” or “return” suggests that an object will be returned as a result of the method execution. Thus, having such methods returning <code>void</code> without documenting where the result is stored is counterintuitive.
Example	The example in Fig. 12 shows the source code of a method named <code>getMethodBodies</code> , defined in class <code>Compiler</code> , which suggests method bodies 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.
Example solution	The example shown in Fig. 12 could be resolved by renaming the method to <code>fillMethodBodies</code> or by adding a documentation: <i>“The method parses the method bodies and stores the result in the parameter unit”</i> .
Developers’ perception	
External	25 participants (89%) perceived this practice as ‘Poor’ or ‘Very Poor’: all agreed that there should be either a renaming (e.g., ‘fill’, ‘parse’, or ‘set’ instead of ‘get’) or code modification (e.g., refactoring or changing the return type). 2 participants (7%) perceived this practice as ‘Neither poor nor good’; 1 participant (4%) perceived this practice as ‘Good’.
	
Internal	For the two examples of this LA that we showed to the internal developers, they would undertake a renaming. For method <code>getTaskAttributes</code> the developer suggested to rename the parameter <code>id2value</code> making it clear that it will hold the result. For method <code>getUpstreamProjects</code> a developer commented: <i>“Some might say that this is OK as it’s a helper method for recursion when building the tree. I wouldn’t”</i> .

```

/**
 * 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(final Object o) {
    return new Object[0];
}

```

Fig. 10 Not implemented condition (B.1).

```

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

```

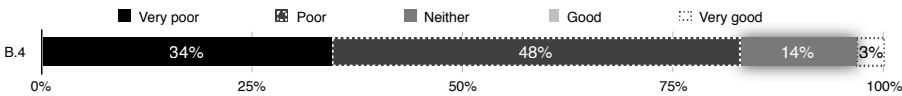
Fig. 11 Validation method does not confirm (B.2).

```

protected void getMethodBodies(
    final CompilationUnitDeclaration unit,
    final 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 < this.parseThreshold) {
        return; // work already done ...
    }
    // real parse of the method...
    this.parser.scanner
        .setSourceBuffer(
            unit.compilationResult.compilationUnit
                .getContents());
    if (unit.types != null) {
        for (int i = unit.types.length; --i >= 0;) {
            unit.types[i].parseMethod(this.parser, unit);
        }
    }
}

```

Fig. 12 “Get” method does not return (B.3).

B.4 - Not answered question	
The method name is in the form of predicate, whereas nothing is returned.	
Rationale	A method whose name is a predicate (e.g., starts with “is”, “has”) is expected to have Boolean as return type where the returned value indicates an assertion or a denial.
Example	Fig. 13 shows an example of method <code>isValid</code> , declared in class <code>ISelectionValidator</code> , where the name suggests a Boolean value as result but nothing is returned.
Consequences	Consequences are similar to those of “ <i>Get</i> ” <i>method does not return</i> . In this case, the developer would even expect to use the method within a conditional control structure, which is however not possible.
Example solution	The example shown in Figure 13 can be resolved by documenting that “ <i>the result of the validation and the validation message are stored in res</i> ”, by changing the return type to Boolean, and by returning <code>true</code> when the selection is valid and <code>false</code> otherwise.
Developers’ perception	
External	24 participants (83%) perceived the practice as ‘Poor’ or ‘Very poor’. Only 4 participants (14%) perceived this practice as ‘Neither poor nor good’ and 3 of them would undertake an action (renaming or code modification). Only 1 participant (3%) perceived it as ‘Very good’ because “ <i>it is understandable</i> ”. This participant indicated C as her main programming languages, while being not expert of Java.
	
Internal	All internal developers perceived this practice as ‘Poor’ or ‘Very poor’. After we pointed out method <code>isSnapped</code> , the code was removed as it was not used anymore. Another developer suggested to rename method <code>isLastWindow</code> .
Causes	<i>Not enough thought.</i>

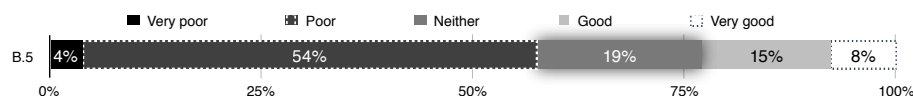
B.5 - Transform method does not return

The method name suggests the transformation of an object, however there is no return value and it is not clear from the documentation where the result is stored.

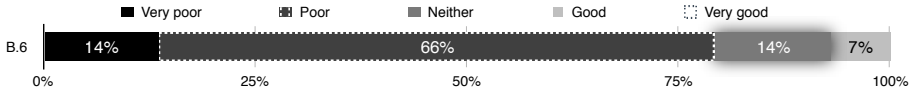
Rationale	A method whose name suggests the transformation of an object is expected to return the result or, if this is not the case, document where the results is stored—e.g., if one of the parameter stores the result then this must be clear from its name/documentation.
Example	An example of this LA is shown in Fig. 14—method <code>javaToNative</code> defined in class <code>LocalSelectionTransfer</code> —where the name suggests a transformation of an object but it is unclear where the result is stored and how to retrieve it.
Consequences	Similar to “ <i>Get</i> ” method does not return. Specifically, here one would expect to be able to assign the result of the method to a variable suggested by the method name (<i>Native</i> in our example, i.e., a platform-specific representation).
Example solution	The example shown in Fig. 14 could document that “ <i>The result of the conversion is stored in <code>transferData</code></i> ” or simply inherit the documentation of the overridden method—as in this case it exists.

Developers’ perception

External	15 participants (58%) perceived this practice as ‘Poor’ or ‘Very poor’. 2 participants justified the expected return type by providing as example the <code>toString</code> method. From the other 11 participants—5 participants (19%) perceived this practice as ‘Neither poor nor good’ and 6 participants (23%) perceived this practice as ‘Good’ or ‘Very Good’—2 would prefer to have a (non <i>void</i>) return type, although perceiving the practice as ‘Neither poor nor good’; and 1 perceived the practice as ‘Very good’ but justified: “ <i>I would blame for anything the superclass as this is a polymorphic method</i> ”. On the contrary, 3 of the participants explicitly stated that no return type should be expected from a transform method.
-----------------	--



Internal	The example we showed to a developer is method <code>PMCamera_Global13dToLocal13d</code> with <code>void</code> return type. The developer decided not to undertake an action “ <i>to save resources—instead of creating a new object and return it, it is convenient to store the result in a parameter.</i> ”
Causes	<i>Developers’ decision.</i>

B.6 - Expecting but not getting a collection	
The method name suggests that a collection should be returned, but a single object or nothing is returned.	
Rationale	A method whose name suggests that a collection is returned is expected to also have a collection as return type. If the method returns a single object then it must be clear from the documentation what is the implicit aggregation function and the method must be considered for renaming.
Example	In the example shown in Fig. 15, the name of the method, defined in class <i>SAXParserBase</i> , 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 Fig. 15, the returned value is completely inconsistent with the method name.
Example solution	A solution for the example shown in Fig. 15 would be to rename the method to <code>isStatisticsEnabled</code> as well as the corresponding attribute.
Developers' perception	
External	23 participants (79%) perceived the practice as 'Poor' or 'Very poor'. To reflect the return type, participants suggested a renaming, e.g., <code>haveStats</code> , <code>statsEnabled</code> , or <code>statsShown</code> . From the other 6 participants, 2 (7%) perceived the practice as 'Good', while 4 (14%) as 'Neither good nor poor'. One of these 4 participants justified the choice after wrongly inferring that <code>stats</code> stands for 'status', whereas another participant was confused by the Boolean return type.
	
Internal	Examples of this LA where developers would undertake a renaming are method <code>getRows</code> returning <code>int</code> where developer suggested <code>getHeight</code> as more appropriate name; method <code>getStates</code> returning <code>State</code> . Examples where developers consider the practice acceptable are method <code>getBounds</code> returning <code>Dimension</code> ; method <code>getValues</code> returning <code>bool</code> where the result is stored in parameter <code>values</code> and the returned value “ <i>indicates success or failure</i> ”.
Causes	<i>Evolution, Not enough thought.</i>

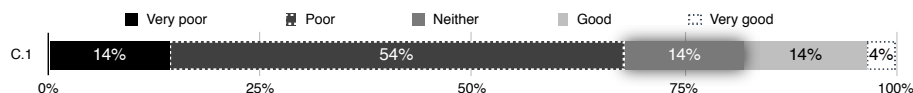
C.1 - Method name and return type are opposite

The intent of the method suggested by its name is in contradiction with what it returns.

Rationale	The name of a method indicates the action that will be performed while its return type specifies the type of the result from this action. As such, the return type must be consistent, i.e., not in contradiction, with the method's name.
Example	The method shown in Fig. 16, defined in class <code>ControlEnableState</code> , is an example of this LA, where the name and return type are inconsistent because the method <code>disable</code> 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).
Example solution	To resolve the example shown in Fig. 16, the class <code>ControlEnableState</code> could be renamed to <code>ControlState</code> to handle the case where the state is enabled but also where the state is disabled. Thus, the inconsistency with method <code>disable</code> is resolved as it will be returning a <code>ControlState</code> .

Developers' perception

External	19 participants (68%) perceived it as ‘Poor’ or ‘Very poor’. From the other 9—4 participants (14%) perceived this practice as ‘Neither poor nor good’ and 5 participants (18%) perceived this practice as ‘Good’ or ‘Very Good’—a participant suggested to rename the return type, to avoid the use of antonyms; another admitted that <i>“Even if the wording is not totally clear we get that it returns the state”</i> . The remaining 7 participants had no issue with this practice, and highlighted that the existing comment <i>“Saves the current enable/disable state ...”</i> is complementary and clarifies the purpose of the method.
-----------------	---



Internal	Regarding method <code>exit_transport_impl</code> returning <code>Enter.Transport</code> , internal developers would rename it, however, they were not certain about the new name <i>“in English there isn't a word (that I know of) which bundles together 'enter' and 'exit'”</i> . Another example of this LA is method <code>player_enemy_impl</code> returning a <code>Player.Ally</code> , where one of the developers justified the decision as part of the design. However, other developers of the system would rename the return type to reflect both states.
Causes	<i>Developers' decision.</i>

```

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

```

Fig. 13 Not answered question (B.4).

```

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

```

Fig. 14 Transform method does not return (B.5).

```

public boolean getStats() {
    return SAXParserBase._stats;
}

```

Fig. 15 Expecting but not getting a collection (B.6).

```

/**
 * Saves the current enable/disable state of the given control
 * and its descendants 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);
}

```

Fig. 16 Method name and return type are opposite (C.1).

C.2 - Method signature and comment are opposite

The documentation of a method is in contradiction with its declaration.

Rationale The leading comment of a method specifies the method's intent at a higher level of abstraction and as such it must be consistent with, i.e., not contradicts, its actual implementation.

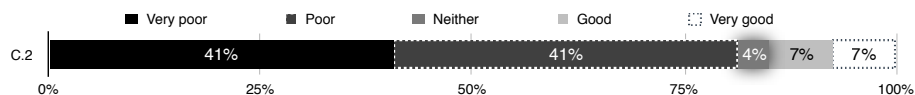
Example Fig. 17 shows method `isNavigateForwardEnabled` where the name of the method is in contradiction with its comment documenting “a back navigation”, as “forward” and “back” are antonyms.

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's signature. Either the one or the other is outdated or inconsistent, and has to be updated.

Example solution The inconsistency in the example shown in Fig. 17 would be resolved by correcting the comment to document “a forward navigation” thus being consistent with the implementation.

Developers' perception

External 22 participants (81%) condemned this practice, with 11 (41%) perceived it as ‘Very poor’. One participant explicitly justified that she would trust the naming rather than the comment. This is also reflected by the high percentage (74%) of participants who perceived that the action to be undertaken would be to change the comment. 1 participants (4%) perceived this practice as ‘Neither poor nor good’; 4 participants (15%) perceived this practice as ‘Good’ or ‘Very Good’.



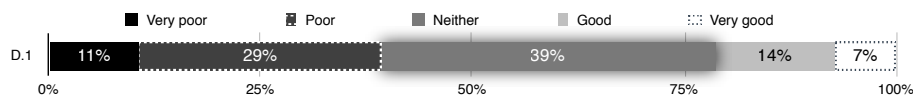
D.1 - Says one but contains many

An attribute name suggests a single instance, while its type suggests that the attribute stores a collection of objects.

Rationale	The name of an attribute indicates what is the object(s) that it contains while the type of an attribute indicates the type of the contained object(s). Thus, there must be a consistency between the name and type, i.e., when the name suggests a single instance the type must also do.
Example	In the example shown in Fig. 18, attribute <code>target</code> of type <code>Vector</code> , it is unclear whether a change affects one or multiple instances in the collection.
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.
Example solution	The inconsistency in the example shown in Fig. 18 can be resolved by renaming the attribute to <code>targetCritics</code> or simply <code>critics</code> .

Developers' perception

External	Only 11 participants (39%) perceived this practice as 'Poor' or 'Very poor'. 11 participants (39%) perceived this LA as 'Neither poor nor good', and 7 of them justified their choice to the lack of context. In other words, whether attribute <code>target</code> of type <code>Vector</code> is a good or poor naming, it depends on whether the target is the entire collection or selected objects contained in the collection. 6 participants (21%) perceived this practice as 'Good' or 'Very good' assuming that <code>target</code> refers to the entire collection.
-----------------	---



Internal	Internal developers suggested renaming for attribute <code>mInstalledPackageInfo</code> of type <code>PackageInfo[]</code> . Regarding attribute <code>projectorImage</code> of type <code>IplImage[]</code> a developer shared “ <i>Could go either way - change or keep. Maybe rename to <code>projectorImagePyramid</code> (because it is one image at different resolutions) but it gets too long.</i> ”. One developer expressed a concern regarding the LA as follows: “ <i>There are technical terms that will most likely sound like plural to an expert of the domain.</i> ”.
-----------------	--

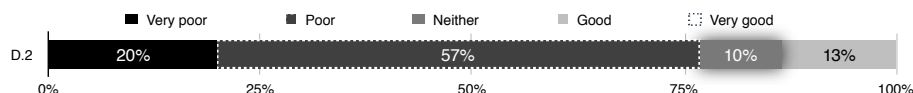
D.2 - Name suggests Boolean but type does not

An attribute name suggests that its value is *true* or *false*, while its declaring type is not Boolean and the declared type and values are not documented.

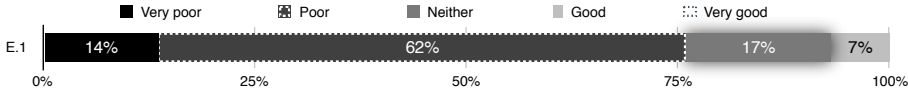
Rationale	The name of an attribute and its type must be consistent in a way that when the name suggests that a Boolean value is contained then the declared type must be indeed Boolean.
Example	Fig. 19 shows one such case defined in class <code>ExceptionHandlerFlowContext</code> . The attribute name— <code>isReached</code> —suggests that the value will be true if something is reached, false otherwise. However, the declaring type is not Boolean.
Consequences	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 Fig. 19, for which the returned type is an array, therefore it is not clear how to handle this attribute.
Example solution	To resolve the inconsistency in the example shown in Fig. 19, the type of the array can be changed to <code>boolean[]</code> or a comment should be added to document how the values are treated, e.g., “0 indicates ‘false’, every other value is treated as ‘true’”.

Developers’ perception

External	23 participants (77%) perceived this practice as at least ‘Poor’ when we showed them attribute <code>isReached</code> of type <code>int[]</code> ; they expected at least an array of Boolean values. A participant suggested <code>reachedItems</code> as a more appropriate name. From the remaining participants, 3 perceived the practice as ‘Neither poor nor good’ (10%) and 4 as ‘Good’ (13%) and assumed values are 0 for <i>false</i> and 1 for <i>true</i> .
-----------------	--



Internal	One questionnaire containing an example of this LA was answered. For attribute <code>_depends</code> of type <code>String</code> , the developer says that the name is well chosen as it matches standards of an imported library. The same developer also find it obvious that the field contains a reference to the packages on which the class depends.
Causes	<i>Developers’ decision.</i>

E.1 - Says many but contains one	
Attribute name suggests multiple objects, but its type suggests a single one.	
Rationale	The name and type of an attribute must be consistent in a way that when the name suggests multiple objects the type must also do. If this is not the case the documentation must state the rationale behind such inconsistency or the attribute must be renamed to include the implicit aggregation function.
Example	In the example shown in Fig. 20, the attribute name, defined in class <code>SAXParserBase</code> , suggests that it contains statistics whereas its type is Boolean.
Consequences	Lack of understanding of the impact of attribute changes (see also <i>Says one but contains many</i>).
Example solution	A solution for the example shown in Fig. 20 would be to rename the attribute to <code>statisticsEnabled</code> .
Developers' perception	
External	22 participants (76%) perceived this practice as 'Poor' or 'Very poor'. 2 of the remaining 7 participants—5 participants (17%) perceived this practice as 'Neither poor nor good' and 2 participants (7%) perceived this practice as 'Good'—suggested that the attribute is a flag indicating whether statistics are enabled. 2 of them also suggested to add comments to improve understandability.
	
Internal	Developers would resolve this LA by changing the implementation for attribute <code>flags</code> of type <code>unsigned char</code> containing multiple bit flags by “ <i>expanding it to become a bitfield</i> ”. An example where developers perceived the “ <i>inconsistency too minor to introduce changes to code working for years</i> ” is attribute named <code>codecs</code> of type <code>ImageCodecInitializer</code> which is an initializer for multiple codecs.
Causes	<i>Not enough thought, reuse, developers' decision.</i>

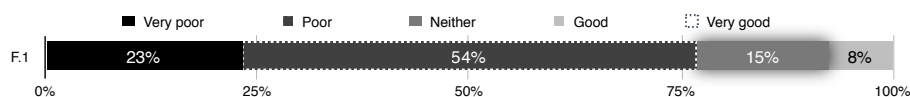
F.1 - Attribute name and type are opposite

The name of an attribute is in contradiction with its type as they contain antonyms.

Rationale	The name and declaring type of an attribute are expected to be consistent with each other and thus one must not contradict the other.
Example	The example of Fig. 21 shows an attribute of class <code>ActionNavigability</code> . The contradiction comes from 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.
Example solution	One way to resolve to inconsistency in the example shown in Fig. 21 would be to rename class <code>MAssociationEnd</code> to <code>MAssociationExtremity</code> . Thus, an object of type <code>MAssociationExtremity</code> called <code>start</code> would mean that the object is the start of the association and will not cause a confusion.

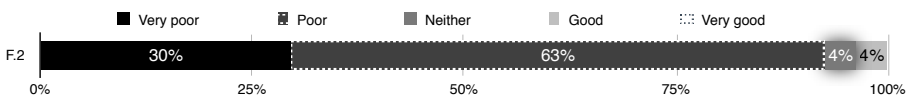
Developers’ perception

External	20 participants (77%) perceived this practice as ‘Poor’ or ‘Very poor’. From the remaining participants, 4 (15%) of them indicated that this naming may or may not be appropriate, based on the context (thus perceiving it as ‘Neither poor nor good’); and 2 (8%) of them perceived this practice as ‘Good’ and believed that the naming is perfectly legitimate (i.e., it is not confusing to deal with “ <i>starting end and finishing end</i> ”) though one recommended comments to clarify this inconsistency.
-----------------	--



Internal	All internal developers perceived this practice as ‘Poor’ or ‘Very poor’. Attribute <code>top_index</code> of type <code>bottom_index</code> is an example of this LA that internal developers would rename.
-----------------	--

F.2 - Attribute signature and comment are opposite	
Attribute declaration is in contradiction with its documentation.	
Rationale	The comment of an attribute clarifies its intent and as such there must be no contradiction between the attribute’s comment and declaration.
Example	The example in Fig. 22 shows an attribute named <code>INCLUDE_NAME_DEFAULT</code> , defined in class <code>EncodeURLTransformer</code> . However, its comment says “ <i>Configuration default exclude pattern</i> ”. Whether the pattern is included or excluded is therefore unclear from the comment and name.
Consequences	A first consequence may be increased comprehension effort as without a deep analysis of the source code, the developer might not clearly understand the role of the attribute. As another risk may be that one simply assumes the intent, i.e., trust the name or the comment, without investigating which of the two is correct.
Example solution	To resolve the inconsistency in Fig. 22, the comment needs to be corrected to document the “ <i>default include pattern</i> ” thus being consistent with the name of the attribute—i.e., <code>INCLUDE_NAME_DEFAULT</code> .

Developers’ perception													
External	<p>A large majority of participants,i.e., 25 participants (93%), perceived this practice as ‘Poor’ or ‘Very poor’. One participant commented: “<i>The most pernicious issue is that most of coders will focus on the meaning of <code>.*@href= .* frame /@src=</code> (whatever it means) although it is of paramount importance to check the ‘exclude/include’ property; depending on coders’ trend to check first the comment or the name of the member!</i>”. Another participant commented: “<i>We don’t know who to believe the comments or the attribute name</i>”. Only 2 participants (7%) were more lenient with their perception—1 participant perceived this practice as ‘Neither poor nor good’ and 1 participant as ‘Good’—one of which commented that one is able to “<i>get the intent</i>”.</p>  <table border="1"><thead><tr><th>Perception</th><th>Percentage</th></tr></thead><tbody><tr><td>Very poor</td><td>30%</td></tr><tr><td>Poor</td><td>63%</td></tr><tr><td>Neither</td><td>4%</td></tr><tr><td>Good</td><td>4%</td></tr><tr><td>Very good</td><td>0%</td></tr></tbody></table>	Perception	Percentage	Very poor	30%	Poor	63%	Neither	4%	Good	4%	Very good	0%
Perception	Percentage												
Very poor	30%												
Poor	63%												
Neither	4%												
Good	4%												
Very good	0%												
Internal	<p>All internal developers perceived this practice as ‘Poor’ or ‘Very poor’. An example where developers believe that a renaming is needed is <code>isOrdered</code> commented as <i>True if the underlying table is <code>BTREE.UNORDERED</code></i>. However, developers believe that no action need to be undertaken for <code>GC.start.time</code> documented as “<i>Time at which we stopped world.</i>” because “<i>stop (the world) is probably synonym to start for GC people</i>”.</p>												

```

/**
 * 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 (this._isForwardEnabled == 1) {
        enabled = true;
    } else {
        if (this._isForwardEnabled != 0) {
            enabled =
                this.navigateForward(false) != null;
        }
    }
    return enabled;
}

```

Fig. 17 Method signature and comment are opposite (C.2).

```

Vector _target;

```

Fig. 18 Says one but contains many (D.1).

```

int[] isReached;

```

Fig. 19 Name suggests Boolean but type does not (D.2).

```

private static boolean _stats = true;

```

Fig. 20 Says many but contains one (E.1).

```

MAssociationEnd start = null;

```

Fig. 21 Attribute name and type are opposite (F.1).

```

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

```

Fig. 22 Attribute signature and comment are opposite (F.2).

4.3.1 Summary of developers' perception on LAs

We can summarize what we have learned from the studies on LAs perceptions as follows:

Does more than what it says (A):

Methods that do more than what they say seem to be perceived acceptable in some situations by both external and internal developers. In such situations the surveyed developers tend to infer the behavior suggested by the name. Sometimes such inference is wrong and may lead to faults. This was the case with the example of *"Is" returns more than a Boolean* (A.2) where 3 of the external developers wrongly mapped the return values for method `isValid` and assumed 2 possible return values whereas in reality there are 3.

Says more than what it does (B):

Developers are less lenient with methods that say more than what they do. LAs from this category are perceived often as unacceptable as the expectations resulting from the method's name/documentation are not fulfilled—as for method `isValid` with `void` return type, example of *Not answered question* (B.4)—or it is unclear how to obtain the result—as for method `getMethodBodies` with `void` return type, example of *"Get" method does not return* (B.3).

Does the opposite (C):

Developers perceive as poor practices methods that do the opposite of what they say. When the inconsistency is between the method's name and return type developers try to infer the actual behavior from the method's comments—which they correctly did for the particular examples that we showed. However, developers are less lenient when the inconsistency is between the method's signature and comments as they wouldn't know which one to trust.

Contains more than what it says (D):

The surveyed developers are more lenient with attributes that contain more than what they say when they feel they need more context—e.g., for *Says one but contains many* (D.1) some developers explained that whether it is a poor practice would depend on the context and in the particular example the attribute declaration is not sufficient to understand the intent. Thus, developers would need to browse the source code and possibly other sources of documentation to clarify the attribute's intent. However developers perceived as poor practice attributes whose *Name suggests Boolean but type does not*. We suspect that examples of LAs in this category may be more likely to increase comprehension effort.

Says more than what it contains (E):

Attributes that say more than what they contain are perceived more severely by external developers; internal developers are more lenient. Thus, although LAs in this category may impede comprehension of newcomers—they may have difficulties to infer the implicit aggregation function—it seems that developers familiar with the code simply get used to and have less issues with those LAs.

Contains the opposite (F):

Attributes that contain the opposite of what they say are perceived as poor practices by the majority of the surveyed developers, especially when the inconsistency

occurs between the attribute’s signature and comments—similar to methods that do the opposite of what they say.

4.4 LAs perceived as particularly poor

Based on the two studies with developers we distill a subset of 11 LAs (see Table 8) that are perceived by external (column **SI**) and/or internal (column **SII**) developers as particularly poor practices. From *Study I*, we consider that LAs are perceived as particularly poor when they are perceived as ‘Poor’ or ‘Very poor’ by at least 75% of the external developers. As proportions for *Study II* are not meaningful—due to the limited number of data points—we consider that LAs are perceived as particularly poor when there is a full agreement among internal developers—i.e., all internal developers perceived them as ‘Poor’ or ‘Very poor’—or when internal developers took an action to resolve them.

There are three LAs that both external and internal developers find particularly unacceptable. Those are LAs concerning the state of an entity (i.e., attributes) and they belong to the “says more than what it does” (B) and “contains the opposite” (F) categories—i.e., *Not answered question* (B.4), *Attribute name and type are opposite* (F.1), and *Attribute signature and comment are opposite* (F.2).

In addition, internal developers appear to be concerned with “*Is*” *returns more than a Boolean* (A.2). External developers are more lenient with this practice as only 60% of them consider it as poor. However, we believe that A.2 may cause comprehension problems as 3 of the external developers that perceive this practice as good wrongly assumed the return values.

Finally, we observe that external developers perceive as particularly unacceptable LAs from all categories.

Table 8 LAs perceived as particularly poor.

		SI	SII
A.2	<i>“Is” returns more than a Boolean</i>		✓
A.3	<i>“Set” method returns</i>	✓	✓
B.2	<i>Validation method does not confirm</i>		✓
B.3	<i>“Get” method does not return</i>	✓	
B.4	<i>Not answered question</i>	✓	✓
B.6	<i>Expecting but not getting a collection</i>	✓	
C.2	<i>Method signature and comment are opposite</i>	✓	
D.2	<i>Name suggests Boolean but type does not</i>	✓	
E.1	<i>Says many but contains one</i>	✓	
F.1	<i>Attribute name and type are opposite</i>	✓	✓
F.2	<i>Attribute signature and comment are opposite</i>	✓	✓

5 Threats to Validity

Threats to *construct validity* concern the relation between theory and observation, and they are mainly related to the accuracy of the measurements we performed to address our research questions. We manually validated the instances of the LAs we showed to the participants, and we selected a representative sample of the different kinds of LAs. Clearly, there is always a risk that the developer’s perception is bound to the particular instance of an LA rather than to its category. However, we limited this threat by collecting comments helping us to understand whether the LAs are indeed a general problem—which we found most of the times to be the case—or whether, instead, it depends on the context.

Regarding the measurement of the participants’ perception, we used Likert scale [Oppenheim, 1992], which helps to aggregate and compare results from multiple participants.

Threats to *internal validity* concern factors that could have influenced our results. When asking participants to evaluate code snippets, we formulate a specific question thus possibly affecting the internal validity of the study as participants may guess the expected answer [Shull et al., 2007]. To cope with this threat we also evaluate a set of examples not containing LAs and show a statistically significant difference in developers’ evaluations. In *Study I*, we analyzed the effect of the experience, the main programming language, and occupation of the participants. Another threat to validity is that external developers are only provided with code snippets and thus unaware of the context, i.e., the particular project that a snippet belongs to. Providing context may lead to more lenient evaluations by external developers as they may resolve the inconsistencies from other places in the code (e.g., from the way the entity is used), which could bias the perception of the practice itself. Also, as participants in *Study I* are external to the project, the lack of domain knowledge may have impacted their perception. We believe that this threat is limited as LAs concern general inconsistencies and thus are mainly domain independent.

Due to the limited number of data points, we did not perform any particular analysis in *Study II*, where we discussed results qualitatively rather than quantitatively. Our results may have been impacted by the fact that participants in *Study II* only validate a subset of the LAs. More data points for each LA may produce different results. More important, we have conducted the two studies to gain insights from different perspectives, i.e., both external and internal developers. A threat for *Study II* is that *internal* developers could have been more lenient with their own code. We mitigated this threat by asking them to motivate their answer and, in any case, also for *Study II* we found a pretty high proportion of poor/very poor perception of LAs.

As shown in Table 3 Note that the majority of the participants are native French speakers and that only for 13% of the participants are native English speakers. However, we believe that this threat to validity is limited as our questions relate to basic grammar rules (e.g., singular/plural) and we analyze the justification for each question to ensure that the participants properly understood the question.

Threats to *external validity* concern the generalizability of our findings. In terms of objects, the two studies have been conducted on three and eight systems respectively. Although we cannot really ensure full diversity [Nagappan et al.,

2013], as explained in Section 3, the chosen systems are pretty different in terms of size and application domain. In terms of subjects, the studies involved both students and professionals (from industry and from the open-source community), as well as developers of projects from which the LAs were detected and developers of other projects.

6 Related Work

This section discusses related work, concerning (i) the relationship between source code lexicon and software quality (Section 6.1), (ii) the identification and analysis of lexicon-related inconsistencies (Section 6.2), and (iii) empirical studies aimed at investigating developers’ perception of code smells (Section 6.3).

6.1 Role of Source code identifiers in Software Quality

Many authors have shown that the quality of the lexicon is an important factor for program comprehensibility.

As discussed in the introduction, *Brooks* [1983] considers identifiers and comments as part of the internal indicators for the meaning of a program.

Shneiderman [1977] presents a syntactic/semantic model of programmer behavior. The syntactic knowledge about a program is built through a perception process; it is precise, language dependent and easily forgettable. The semantic knowledge is built through cognition; it is language independent and concerns important concepts at different levels of details. On the basis of several experiments, *Shneiderman and Mayer* [1975] observed a significantly better program comprehension by subjects with commented programs. Higher number of subjects located bugs in commented programs compared to not commented programs, although the difference is not statistically significant. They argue that program comments and mnemonic identifiers simplify the conversion process from the program syntax to the program internal semantic representation.

Chaudhary and Sahasrabudde [1980] argue that the psychological complexity of a program—i.e., the characteristics that make a program difficult to be understood—is an important aspect of program quality. They identify several features that contribute to the psychological complexity one of which is termed “meaningfulness”. They argue that meaningful variable names and comments facilitate program understanding as they facilitate the relation between the program semantics and the problem domain. An experiment with students using different versions of FORTRAN programs—with and without meaningful names—confirms the hypothesis.

Weissman [1974a] also considers that the program form—e.g., comments, choice of identifiers, paragraphing—is an important factor that affects program complexity, in particular suggesting that meaningless and incorrect comments can be harmful and that mnemonic names of reasonable length ease program understanding. However empirical evidence showed that comments lead to faster but more error prone hand simulation of a program [*Weissman*, 1974b]. *Sheil* [1981] reviews studies on the psychological research on programming, and argues that the in-

effectiveness of the research in the domain is partly due to the unsophisticated experimental techniques.

Other authors have focused their attention on source code identifiers and their importance for various tasks in software engineering. Among them, *Caprile and Tonella* [1999, 2000], *Merlo et al.* [2003], and *Anquetil and Lethbridge* [1998] show that identifiers carry important source of information and that identifiers are often the starting point for program comprehension. *Deissenbock and Pizka* [2005] provided guidelines for the production of high-quality identifiers. Later, *Laurie et al.* [2006, 2007] performed an empirical study to assess the quality of source code identifiers, and suggest that the identification of words composing identifiers could contribute to a better comprehension.

6.2 Identifying inconsistencies in the lexicon

Previous studies have shown that poor source code lexicon correlates with faults [*Abebe et al.*, 2012], and negatively affects concept location [*Abebe et al.*, 2011].

Abebe and Tonella [2011] extract concepts and relations between concepts from program identifiers to build an ontology. They use the ontology to help developers in choosing identifiers consistent with the concepts already used in the system [*Abebe and Tonella*, 2013]. To this aim, given partially written identifiers, they suggest and rank candidate completions and replacements. We complement the above work, as Abebe and Tonella focus on the quality of the lexicon, whereas we identify inconsistencies among identifiers, source code, and comments.

De Lucia et al. [2011] proposed an approach and tool—named COCONUT—to ensure consistency between the lexicon of high-level artifacts and of source code. In their approach, the inconsistent lexicon is measured in terms of textual similarity between high-level artifacts traced to the code, and the code itself. In addition, COCONUT uses the lexicon of high-level artifacts to suggest appropriate identifiers.

Tan et al. [2007, 2011, 2012] proposed several approaches to detect inconsistencies between code and comments. Specifically, @ICOMMENT [*Tan et al.*, 2007] detects lock- and call-related inconsistencies; the validation made by developers confirmed 19 of the detected inconsistencies. @ACOMMENT [*Tan et al.*, 2011] detects synchronization inconsistencies related to interrupt context, and the evaluation by developers confirmed 7 previously unknown bugs. @TCOMMENT infers properties from *Javadoc* related to null values and exceptions; then, it generates tests cases by searching for violations of the inferred properties. Also in this case, *Tan et al.* reported the detected inconsistencies to the developers who indeed resolved 5 of them. *Zhong et al.* [2011] automatically generate specifications from API documentation concerning resource usage, namely creation, lock, manipulation, unlock, and closure. They contacted developers of the open-source projects who confirmed 5 previously unknown defects.

While the approaches described above address inconsistencies specific to certain source code aspect/implementation technology—i.e., lock/call, null values/exceptions, synchronization, and resource usage—our approach can be considered as complementary as it deals with generic naming and commenting issues that can arise in OO code, and specifically in the lexicon and comments of methods and attributes.

6.3 Developers' perception of code smells

Yamashita and Moonen [2013] performed a study—involving 85 professionals—with the aim of investigating the perception of code smells, in particular, the degree of awareness of code smells, their severity, and the usefulness of automatic tool support. Surprisingly, 23 of the participants (32%) were not aware of such code smells. From the remaining 50 participants, i.e., those that have at least heard of anti-patterns and code smells, only 3 participants (6%) were not concerned about the presence of code smells. 47 of the participants (94%) were concerned at a different level—10 (20%) were slightly concerned, 11 (22%) were somewhat concerned, 19 (38%) were moderately concerned, and 7 (14%) were extremely concerned. Yamashita and Moonen performed categorical regression analysis and found that the more familiar participants are with anti-patterns and code smells, the more concerned they are. *Palomba et al.* [2014] also studied developers' perceptions of code smells. They evaluated examples of 12 code smells found in 3 open-source Java projects from the perspective of 34 external and internal developers. Their results show that there are some code smells that developers do not perceive as poor practices. They also observed that for several code smells experienced developers are more concerned than less experienced developers.

We share with the above works the interest in how developers perceive poor practices. The main difference between previous work and our work is that while they evaluate practices that have been out there more than a decade, we study practices with which developers were not at all familiar with *Study I* or just introduced to *Study II*. This could be one of the reasons why a lower number of participants perceive LAs as 'Poor' or 'Very Poor'—69% and 51% for *Study I* and *Study II* respectively—as opposed to anti-patterns and code smells—94% when only considering participants familiar with anti-patterns and code smells. Also, while they evaluate the awareness and the concern of professionals about the code smells in general, we focus on evaluating the perception of developers of LAs i) through the mean of concrete examples thus allowing us to also investigate possible solutions, and ii) from both perspectives i.e., newcomers (*Study I*), and internal developers (*Study II*).

7 Conclusion and Future Work

This work aimed at investigating the developers' perception of Linguistic Anti-patterns (LAs)—i.e., “poor practices in the naming, documentation, and choice of identifiers in the implementation of an entity, thus possibly impairing program understanding”—and the extent to which they suggest that such LAs need to be removed. The studies concerned a catalog of 17 types of LAs—defined in our previous work [Arnaoudova et al., 2013]—that we conjecture to be poor practices. In this paper, we rely on the opinion of developers as an indication of the quality of source code containing such poor practices with the aim of confirming or refuting our conjecture.

First, we conducted a study involving 30 *external* developers among graduate students and professional, i.e., people that did not participate to the development of the system in which the LAs were detected and unaware of the notion of LAs. They provided information about their perception of LAs found in 3 Java open-

source projects, and the majority of them (69%) indicated that such LAs are poor or very poor practices. Overall, developers perceived as more serious ones the instances where the inconsistency involved both method signature and comments. The perception of external developers is important as 1) it provides an indication of the difficulties that newcomers may encounter understanding code that contains LAs, and 2) it is an unbiased opinion.

In a second study we asked 14 (*internal*) developers of 7 open-source Java/C++ projects and one C++ commercial system to provide us their perception of LAs we found in the code of their projects. Internal developers provides us 1) with an indication whether code containing LAs is problematic even for people that are familiar with the project and 2) with insights on why LAs occurred in the code and how can they be refactored. 51% of respondents evaluated LAs as poor or very poor practices. The percentage is lower compared to the one observed with external developers, as in some cases internal developers perceive LAs acceptable in the particular context. When asked why the LAs were possibly introduced—and developers had elements to answer—they pointed out maintenance activities—e.g., done by developers different from the original code authors—that deteriorated the lexicon quality, or lack of attention to naming conventions/comments. For a conspicuous proportion of LAs (56%) developers highlighted that such LAs should be removed and, at the time of writing this paper, internal developers had already resolved 10% of the cases containing LAs that we pointed out. As a result of the studies with developers, we distill a subset of LAs i) that are perceived as poor practices by at least 75% of the *external* developers, ii) that are perceived as poor practices by all *internal* developers, or iii) for which *internal* developers took an action to remove it. There are three LAs that both external and internal developers agree on and perceived as particularly poor. Those are LAs concerning the state of an entity (i.e., attributes) and they belong to the “says more than what it does” (B) and “contains the opposite” (F) categories—i.e., *Not answered question* (B.4), *Attribute name and type are opposite* (F.1), and *Attribute signature and comment are opposite* (F.2). External developers found particularly unacceptable (i.e., more than 80% of them perceived as poor or very poor) the LAs with a clear dissonance between code behavior and its lexicon—i.e., *“Get” method does not return* (B.3), *Not answered question* (B.4), *Method signature and comment are opposite* (C.2), and *Attribute signature and comment are opposite* (F.2). Given the extremely high level of agreement on those LAs, our results encourage the use of a recommender tool highlighting LAs, like such as the Checkstyle extension we developed and described in Section 2.

Clearly, one must consider that, whether or not developers could remove LAs also depends on the impact that this can have on the whole system. In other words, developers are less prone to remove LAs if this has a large impact on the code, as such change can be too risky. Instead, it can be more useful to point out LAs as developers write source code—e.g., on-the-fly using our LINGUISTIC ANTIPATTERN DETECTOR (LAPD) CHECKSTYLE PLUGIN—thus removing or limiting the impact on other code entities.

Work-in-progress includes: (i) proposing automatic refactorings to resolve LAs, and (ii) performing a study involving developers using (or not) the LAPD CHECKSTYLE PLUGIN, with the aim of observing to what extent the recommendations will be followed, and to what extent will the code lexicon be improved.

Acknowledgements The authors would like to thank the participants to the two studies for their precious time and effort. They made this work possible.

Appendices

A Detection

A.1 - “Get” - more than an accessor: Find accessor methods by identifying methods whose name starts with ‘get’ and ends with a substring that corresponds to an attribute in the same class and where the attribute’s declared type and the accessor’s return type are the same. Then, identify those accessors that are performing more actions than returning the corresponding attribute. 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. For a detection built on top of an Abstract Syntax Tree (AST) expressions other than a return statement—where the attribute is returned—can be allowed only if they are child of a conditional check for null value. Other measures for complexity, such as LOC or McCabe’s Cyclomatic Complexity, can be used for a simpler but less accurate detection.

A.2 - “Is” returns more than a Boolean: Find methods starting with “is” whose return type is not Boolean.

A.3 - “Set” method returns: Find modifier methods (or more generally methods whose name starts with “set”) and whose return type is different from void.

A.4 - Expecting but not getting a single instance: Find methods returning a collection (e.g., array, list, vector, etc.) but whose name ends with a singular noun and does not contain a word implying a collection (e.g., array, list, vector, etc.).

B.1 - Not implemented condition: 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).

B.2 - Validation method does not confirm: Find validation methods (e.g., method names starting with “validate”, “check”, “ensure”) whose return type is void and that do not throw an exception.

B.3 - “Get” method does not return: Find methods where the name suggests a return value (e.g., names starting with “get”, “return”) but where the return type is void.

B.4 - Not answered question: Find methods whose name is in the form of predicate (e.g., starts with “is”, “has”) and whose return type is void.

B.5 - Transform method does not return: Find methods whose name suggests a transformation of an object, (e.g., `toSomething`, `source2target`) but its return type is void.

B.6 - Expecting but not getting a collection: 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.

C.1 - Method name and return type are opposite: Find methods where the name and return type contain antonyms.

C.2 - Method signature and comment are opposite: Find methods whose name or return type have an antonym relation with its comment.

D.1 - Says one but contains many: Find attributes having a name ending with a singular noun and having a collection as declaring type.

D.2 - Name suggests Boolean but type does not: 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/present participle, but whose declaring type is not Boolean.

E.1 - Says many but contains one: Find attributes having a name ending with a plural noun, however their type is not a collection neither it contains a plural noun.

F.1 - Attribute name and type are opposite: Find attributes whose name and declaring type contain antonyms.

F.2 - Attribute signature and comment are opposite: Find attributes whose name or declaring type have an antonym relation with its comment.

References

- Abbes, M., F. Khomh, Y.-G. Guéhéneuc, and G. Antoniol, An empirical study of the impact of two antipatterns, Blob and Spaghetti Code, on program comprehension, in *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR)*, pp. 181–190, 2011.
- Abebe, S., and P. Tonella, Towards the extraction of domain concepts from the identifiers, in *Proceedings of the Working Conference on Reverse Engineering (WCRE)*, pp. 77–86, 2011.
- Abebe, S., and P. Tonella, Automated identifier completion and replacement, in *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR)*, pp. 263–272, 2013.
- Abebe, S. L., S. Haiduc, P. Tonella, and A. Marcus, The effect of lexicon bad smells on concept location in source code, in *Proceedings of the International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pp. 125–134, 2011.
- Abebe, S. L., 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)*, pp. 235–244, 2012.
- Anquetil, N., and T. Lethbridge, Assessing the relevance of identifier names in a legacy software system, in *Proceedings of the International Conference of the Centre for Advanced Studies on Collaborative Research (CASCON)*, pp. 213–222, 1998.
- Arnaoudova, V., M. Di Penta, G. Antoniol, and Y.-G. Guéhéneuc, A new family of software anti-patterns: Linguistic anti-patterns, in *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR)*, pp. 187–196, 2013.
- Arnaoudova, V., L. Eshkevari, M. Di Penta, R. Oliveto, G. Antoniol, and Y.-G. Guéhéneuc, Repent: Analyzing the nature of identifier renamings, *IEEE Transactions on Software Engineering (TSE)*, 40(5), 502–532, 2014.
- Brooks, R., Towards a theory of the comprehension of computer programs, *International Journal of Man-Machine Studies*, 18(6), 543–554, 1983.

- Brown, W. J., R. C. Malveau, W. H. Brown, H. W. McCormick III, and T. J. Mowbray, *Anti Patterns: Refactoring Software, Architectures, and Projects in Crisis*, 1st ed., John Wiley and Sons, 1998a.
- Brown, W. J., R. C. Malveau, H. W. M. III, and T. J. Mowbray, *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*, John Wiley & Sons, Inc., 1998b.
- Caprile, B., and P. Tonella, Nomen est omen: Analyzing the language of function identifiers, in *Proceedings of Working Conference on Reverse Engineering (WCRE)*, pp. 112–122, 1999.
- Caprile, B., and P. Tonella, Restructuring program identifier names, in *Proceedings of the International Conference on Software Maintenance (ICSM)*, pp. 97–107, 2000.
- Chaudhary, B. D., and H. V. Sahasrabuddhe, Meaningfulness as a factor of program complexity, in *Proceedings of the ACM Annual Conference*, ACM '80, pp. 457–466, ACM, 1980.
- De Lucia, A., M. Di Penta, and R. Oliveto, Improving source code lexicon via traceability and information retrieval, *IEEE Transactions on Software Engineering*, 37(2), 205–227, 2011.
- Deissenbock, F., and M. Pizka, Concise and consistent naming, in *Proceedings of the International Workshop on Program Comprehension (IWPC)*, pp. 97–106, 2005.
- Fowler, M., *Refactoring: improving the design of existing code*, Addison-Wesley, 1999.
- Gamma, E., R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object Oriented Software*, Addison-Wesley, Boston, MA, USA, 1995.
- Glaser, B. G., *Basics of grounded theory analysis*, Sociology Press, 1992.
- Grissom, R. J., and J. J. Kim, *Effect sizes for research: A broad practical approach*, 2nd edition ed., Lawrence Earlbaum Associates, 2005.
- Groves, R. M., F. J. Fowler Jr., M. P. Couper, J. M. Lepkowski, E. Singer, and R. Tourangeau, *Survey Methodology*, 2nd edition, Wiley, 2009.
- Hintze, J. L., and R. D. Nelson, Violin plots: A box plot-density trace synergism, *The American Statistician*, 52(2), 181–184, 1998.
- Jedlitschka, A., and D. Pfahl, Reporting guidelines for controlled experiments in software engineering, in *International Symposium on Empirical Software Engineering*, 2005.
- Khomh, F., M. Di Penta, and Y.-G. Guéhéneuc, An exploratory study of the impact of code smells on software change-proneness, in *Proceedings of the Working Conference on Reverse Engineering (WCRE)*, pp. 75–84, 2009.
- Khomh, F., M. Di Penta, Y.-G. Guéhéneuc, and G. Antoniol, An exploratory study of the impact of antipatterns on class change- and fault-proneness, *Empirical Software Engineering*, 17(3), 243–275, 2012.
- Kitchenham, B., S. Pfleeger, L. Pickard, P. Jones, D. Hoaglin, K. El Emam, and J. Rosenberg, Preliminary guidelines for empirical research in software engineering, *IEEE Transactions on Software Engineering (TSE)*, 28(8), 721–734, 2002.
- Lawrie, D., C. Morrell, H. Feild, and D. Binkley, What's in a name? a study of identifiers, in *Proceedings of the International Conference on Program Comprehension (ICPC)*, pp. 3–12, 2006.

-
- Lawrie, D., C. Morrell, H. Feild, and D. Binkley, Effective identifier names for comprehension and memory, *Innovations in Systems and Software Engineering*, 3(4), 303–318, 2007.
- Merlo, E., I. McAdam, and R. De Mori, Feed-forward and recurrent neural networks for source code informal information analysis, *Journal of Software Maintenance*, 15(4), 205–244, 2003.
- Miller, G. A., WordNet: A lexical database for English, *Communications of the ACM*, 38(11), 39–41, 1995.
- Moha, N., Y.-G. Guéhéneuc, L. Duchien, and A.-F. Le Meur, DECOR: A method for the specification and detection of code and design smells, *IEEE Transactions on Software Engineering (TSE'10)*, 36(1), 20–36, 2010.
- Nagappan, M., T. Zimmermann, and C. Bird, Diversity in software engineering research, in *Proceedings of the joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pp. 466–476, 2013.
- Oppenheim, A. N., *Questionnaire Design, Interviewing and Attitude Measurement*, Pinter, London, 1992.
- Palomba, F., G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk, Detecting bad smells in source code using change history information, in *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pp. 268–278, 2013.
- Palomba, F., G. Bavota, M. D. Penta, R. Oliveto, and A. D. Lucia, Do they really smell bad? a study on developers' perception of code bad smells, in *International Conference on Software Maintenance and Evolution (ICSME)*, p. to appear, 2014.
- Parsons, J., and C. Saunders, Cognitive heuristics in software engineering: Applying and extending anchoring and adjustment to artifact reuse, *IEEE Transactions on Software Engineering (TSE)*, 30(12), 873–888, 2004.
- Prechelt, L., B. Unger-Lamprecht, M. Philippsen, and W. Tichy, Two controlled experiments assessing the usefulness of design pattern documentation in program maintenance, *IEEE Transactions on Software Engineering (TSE)*, 28(6), 595–606, 2002.
- Rațiu, D., S. Ducasse, T. Girba, and R. Marinescu, Using history information to improve design flaws detection, in *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR)*, pp. 223–232, 2004.
- Sheil, B. A., The psychological study of programming, *ACM Computing Surveys (CSUR)*, 13(1), 101–120, 1981.
- Shneiderman, B., Measuring computer program quality and comprehension, *International Journal of Man-Machine Studies*, 9(4), 465–478, 1977.
- Shneiderman, B., and R. Mayer, Towards a cognitive model of programmer behavior, *Tech. Rep. 37*, Indiana University, 1975.
- Shull, F., J. Singer, and D. I. Sjøberg, *Guide to Advanced Empirical Software Engineering*, Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007.
- Strauss, A. L., *Qualitative analysis for social scientists*, Cambridge University Press, 1987.
- Takang, A., P. A. Grubb, and R. D. Macredie, The effects of comments and identifier names on program comprehensibility: an experiential study, *Journal of Program Languages*, 4(3), 143–167, 1996.

-
- Tan, L., D. Yuan, G. Krishna, and Y. Zhou, `/*iComment: bugs or bad comments?*/`, *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, 41(6), 145–158, 2007.
- Tan, L., Y. Zhou, and Y. Padioleau, `acomment`: Mining annotations from comments and code to detect interrupt related concurrency bugs, in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2011.
- Tan, S. H., 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)*, pp. 260–269, 2012.
- Torchiano, M., Documenting pattern use in Java programs, in *Proceedings of the International Conference on Software Maintenance (ICSM)*, pp. 230–233, 2002.
- Toutanova, K., 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)*, pp. 63–70, Association for Computational Linguistics, 2000.
- Weissman, L., Psychological complexity of computer programs: An experimental methodology, *SIGPLAN Not.*, 9(6), 25–36, 1974a.
- Weissman, L. M., A methodology for studying the psychological complexity of computer programs., Ph.D. thesis, 1974b.
- Wohlin, C., P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in Software Engineering - An Introduction*, Kluwer Academic Publishers, 2000.
- Woodfield, S. N., H. E. Dunsmore, and V. Y. Shen, The effect of modularization and comments on program comprehension, in *Proceedings of the International Conference on Software Engineering (ICSE)*, pp. 215–223, 1981.
- Yamashita, A., and L. Moonen, Do developers care about code smells? - an exploratory survey, in *Proceedings of the Working Conference on Reverse Engineering (WCRE)*, pp. 242–251, 2013.
- Zhong, H., L. Zhang, T. Xie, and H. Mei, Inferring specifications for resources from natural language api documentation, *Automated Software Engineering*, 18(3-4), 227–261, 2011.



Venera Arnaoudova is a research associate at Polytechnique Montréal (Canada). She received her bachelor degree in computer and electrical engineering (major of computer science) from the engineering school Polytech'Lille (France) and her master degree in computer science from Concordia University (Canada) in 2008. She received her Ph.D. degree in 2014 from Polytechnique Montréal under the supervision of Dr. Giuliano Antoniol and Dr. Yann-Gaël Guéhéneuc. Her research interest is in the domain of software evolution and particularly, the analysis of source code lexicon and documentation, empirical software engineering, refactoring, patterns, and antipatterns. Her dissertation focused on the improvement of the code lexicon and its consistency using natural language processing, fault prediction models, and empirical studies. More information available at: <http://www.veneraarnaoudova.ca/>.



Massimiliano Di Penta is associate professor at the University of Sannio, Italy since December 2011. Before that, he was assistant professor in the same University since December 2004. His research interests include software maintenance and evolution, mining software repositories, empirical software engineering, search-based software engineering, and service-centric software engineering. He is currently involved as principal investigator for the University of Sannio in a European Project about code search and licensing issues (MARKOS - www.markosproject.eu). Previously, he was principal investigator in other national and European projects on topics related to software evolution and service-centric software engineering. He is author of over 190 papers appeared in international journals, conferences and workshops. He serves and has served in the organizing and program committees of over 100 conferences such as ICSE, FSE, ASE, ICSM, ICPC, GECCO, MSR WCRE, and others. He has been general co-

chair of various events, including the 10th IEEE Working Conference on Source Code Analysis and Manipulation (SCAM 2010), the 2nd International Symposium on Search-Based Software Engineering (SSBSE 2010), and the 15th Working Conference on Reverse Engineering (WCRE 2008). Also, he has been program chair of events such as the 28th IEEE International Conference on Software Maintenance (ICSM 2012), the 21st IEEE International Conference on Program Comprehension (ICPC 2013), the 9th and 10th Working Conference on Mining Software Repository (MSR 2013 and 2012), the 13th and 14th Working Conference on Reverse Engineering (WCRE 2006 and 2007), the 1st International Symposium on Search-Based Software Engineering (SSBSE 2009), and other workshops. He is currently member of the steering committee of ICSME, MSR, SSBSE, and PROMISE. Previously, he has been steering committee member of other conferences, including ICPC, SCAM, and WCRE. He is in the editorial board of IEEE Transactions on Software Engineering, the Empirical Software Engineering Journal edited by Springer, and of the Journal of Software: Evolution and Processes edited by Wiley.



Giuliano Antoniol (Giulio) received his Laurea degree in electronic engineering from the Università di Padova, Italy, in 1982. In 2004 he received his PhD in Electrical Engineering at Polytechnique Montréal. He worked in companies, research institutions and universities. In 2005 he was awarded the Canada Research Chair Tier I in Software Change and Evolution. He has participated in the program and organization committees of numerous IEEE-sponsored international conferences. He served as program chair, industrial chair, tutorial, and general chair of international conferences and workshops. He is a member of the editorial boards of four journals: the Journal of Software Testing Verification & Reliability, the Journal of Empirical Software Engineering and the Software Quality Journal and the Journal of Software Maintenance and Evolution: Research and Practice. Dr. Giuliano Antoniol served as Deputy Chair of the Steering Committee

for the IEEE International Conference on Software Maintenance and Evolution. He contributed to the program committees of more than 30 IEEE and ACM conferences and workshops, and he acts as referee for all major software engineering journals. He is currently Full Professor at the Polytechnique Montreal, where he works in the area of software evolution, software traceability, search based software engineering, software testing and software maintenance.