Finding and Fixing Java Naming Bugs with the Lancelot Eclipse Plugin

Edvard K. Karlsen
Sør-Trøndelag University College, Norway
edvardkk@stud.hist.no

Einar W. Høst
Computas AS, Norway
einarwh@gmail.com

Bjarte M. Østvold
Norwegian Computing Center, Norway
bjarte@nr.no

Abstract

The Lancelot plugin extends the integrated development environment Eclipse with support for finding and fixing 'naming bugs' in Java programs. A naming bug is a mismatch between the name and implementation of a method, in the sense that the pairing of name and implementation do not correspond to the implicit method naming conventions used by many well-known open source applications.

Lancelot has not been presented before, but its theoretical foundations and evaluation have been published [4]. The contribution of the present paper is to present a publicly available tool building on our theory, explain the design of the tool, including some necessary adaptations to the interactive use setting, and report on our experience with it. The source code of Lancelot is available under an open source license.

Categories and Subject Descriptors D.2.6 [*Programming Environments*]: Interactive environments; D.2.5 [*Testing and Debugging*]: Debugging aids; F.3.2 [*Semantics of Programming Languages*]: Program analysis

General Terms Design, Languages

Keywords Naming bugs, Eclipse, Java

1. Introduction

We present Lancelot, a plugin that extends the state-of-the-art integrated development environment Eclipse with support for finding and fixing 'naming bugs' in Java programs. Previously, we have published on the theory of naming bugs and put the theory to use to find such bugs in many open source Java applications [4, 3]. The contribution of the present paper is to present a publicly available tool building on our theory, explain the design of the tool, including some necessary adaptations to the setting of interactive use, and report on our experience with its use. The Lancelot web-site is `http://code.google.com/p/lancelot-eclipse/`.

The next two sub-sections explain what we mean by 'good names' and what naming bugs are. Then, Section 2 presents the application-independent analysis required before using Lancelot on a concrete Java application. Section 3 explains the inner working of Lancelot, specifically how it finds and fixes naming bugs. Section 4 shows Lancelot in use on a real Java application and Section 5 discusses future work on the plugin. Appendix A explains how to get hold of Lancelot, both as a binary for Eclipse and as source code.

1.1 Good names

Industrial practitioners argue for using good names in programming [1, 8, 9] and there is research showing that identifier quality affects our ability to comprehend programs [6]. Yet programmers looking for guidance can do little better than looking at naming convention documents, such as those provided for Java. Here is a typical quote for the kind of advice in such documents: "Except for variables, all instance, class, and class constants are in mixed case with a lowercase first letter."[1] There are tools that help programmers be lexically consistent with these and other conventions. There has, however, not been any tools that help programmers write *meaningful* identifiers. Lancelot is such a tool.

An identifier represents some program entity. We may use the identifier to refer to that entity, and thus the identifier is also an abstraction of the entity. The meaning we as programmers assign to the identifier is connected to the meaning of the entity, that is, its implementation. As programmers we also expect the abstraction to be sound: the identifier must be a suitable replacement for the entity. So we have two criteria for a good name: First, the name should be *meaningful*, that is, a programmer can to some extent understand a program entity from its name. Second, the name should be *appropriate* for the program entity in question, that is, it should faithfully describe the named entity and not be misleading.

1.2 Naming bugs

Consider the following example, taken from AspectJ 1.6.11, where the method name has been replaced by underscores:

```
/**
 * @return field object with given name, or null
 */
public Field ___(String name) {
  for (Iterator e = this.field_vec.iterator();
       e.hasNext();) {
    Field f = (Field) e.next();
    if (f.getName().equals(name))
      return f;
  }
  return null;
}
```

[1] `http://www.oracle.com/technetwork/java/codeconventions-135099.html`

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PEPM'12, January 23–24, 2012, Philadelphia, PA, USA.
Copyright © 2012 ACM 978-1-4503-1118-2/12/01…$10.00

Most Java programmers will be able to give a name to this method: clearly, this is a *find* method. We would probably name it `findField` since it is a method that tries to find a `Field`. In AspectJ, however, the name used is `containsField`. This is an example of what we call a *naming bug*, since the name is not appropriate. The name `containsField` is a question that expects a Boolean reply ("Do you [the object] contain a field with this name?"), rather than an instruction to return an object ("Find me the field with this name.").

Other kinds of naming bugs are unintelligible method names (`frobNitz`, `foo`) and too generic method names (`doThings`). A programmer cannot assign meaning to such identifiers as their names are meaningless. Good names, however, must be meaningful.[2]

2. Rule book generation

This section explains how we build rules for checking Java programs for naming bugs, drawing on our previous work [4]. Our underlying assumption is that good names are derivable from a software corpus of 100 well-known Java applications [4, Table 7] with over one million methods. Consider the three-phase process depicted in Figure 1. Rule book generation consists of phase one and phase two; the third phase, finding naming bugs, is explained later in Section 3.

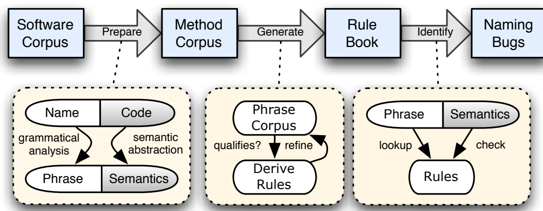

Figure 1. Overview of the naming bug checking process.

In the first phase (Section 2.1) we transform the software corpus into a method corpus, where method names and method implementations are replaced by idealised versions. In the second phase (Section 2.2) we make naming rules by examining the semantics of methods with similar names. The third phase (Section 3) involves checking a specific Java application for naming bugs using Lancelot. This means looking for methods that break the naming rules.

2.1 Building the method corpus

In the method corpus names and method implementations are replaced by idealised versions. First, we analyse the method name, splitting it into individual words and doing a part-of-speech analysis [7] to tag those words with information about their grammatical role in the name. That is, we do natural language analysis of the method name, and tag individual words with their role in the 'name sentence'. A tag is either a grammatical element ('verb', 'noun', 'adjective', 'adverb', etc.) or one of the special tags: 'type' signifying a Java type in the scope of the method, 'number' signifying a numeric value, or 'unknown' signifying that the analysis was unable to tag that word. Tag candidates for individual words are found using WordNet [2] and a hand-made dictionary of programming-specific terms. In the corpus of idealised methods, each method name, for example, the name `findField`, is replaced by a list of tagged words like '`find-field`' with tags 'verb' and 'type'. This allows us to abstract names into *phrases* by replacing one or more words in a name by tags or by the wildcard symbol '*'. For example, the phrase '`is-*`' is more abstract than the phrase '`is-`⟨adjective⟩', which again is more abstract than the name '`is-empty`'. We also say that a more general phrase *matches* a more specific phrase or a name.

Second, we analyse the method implementation, that is, the method signature and its Java bytecode. Comparing regular method implementations is infeasible[3], so in the method corpus we replace each method implementation with a vector of binary numbers, called a *semantic profile*. Corresponding to each bit position of the vector is a *property*. A bit is set in the semantic profile of a method m if m has the corresponding property. The value of each property is determined by method-local static analysis. A semantic profile is an abstraction of a method's behaviour, and comparing method implementations via their semantic profiles is simple. The list of properties used in semantic profiles appear in Table 1. The choice of properties is based on our knowledge of Java programming.

Signature	
Returns void*	Returns reference
Returns int	Returns boolean
Returns string	No parameters*
Return type in name	Parameter type in name
Data Flow	
Reads field*	Writes field*
Writes parameter value to field	Returns field value
Returns created object	Runtime type check*
Object Creation	
Creates regular objects*	Creates string objects
Creates custom objects	Creates own type objects
Control Flow	
Contains loop*	Contains branch
Multiple return points*	
Exception Handling	
Throws exceptions*	Catches exceptions*
Exposes checked exceptions	
Method Call	
Recursive call*	Same name call*
Same verb call*	Method call on field value
Method call on parameter value	Parameter value passed to method call on field value

Table 1. Properties divided into groups. Orthogonal properties in a group are marked with an asterisk.

2.2 Creating the rule book

Naming rules for good names are derived for each prevalent phrase in the method corpus. Since a phrase can match several methods, the number of prevalent phrases is larger than the number of prevalent method names, allowing us to make more rules than if we were to consider regular names. Prevalent phrases become part of a corpus 'rule book' together with rules for good use of the phrase. The intuition is that all methods whose name are matched by a phrase must obey the corresponding rules.

We divide the method corpus into sub-corpora corresponding to non-overlapping phrases. Looking at one such phrase corpus, and one property, we find the frequency values for the property by considering each semantic profile in that corpus. The intuition of this frequency value is that it gives the probability that the property holds for methods in the phrase corpus. A frequency value close to 0 means that it is rare for methods in the phrase corpus to have

[2] Note that these degenerate names are not in any way excluded from our analysis.

[3] One of several problems with comparing regular method implementations is that equality is not generally decidable. Also, we want a coarser notation of equality than semantic identity in order to effectively group and compare implementations.

the corresponding property, and a value close to 1 means that it is common for the methods to have the property. Thus, each rule requires either inclusion or omission of a specific property. From this we make rules for each phrase-property combination. The *rule set* of a phrase is the set of all rules found for all properties in Table 1.

An example of a rule is that a method whose name is matched by the phrase '`contains`-*' should have the property 'Returns boolean'. This rule was broken by the AspectJ method implementation shown in Section 1.2.

To illustrate how to generate a rule book, consider this contrived mini-corpus:

$$\{(\text{'find'}, 100), (\text{'find-field'}, 101), (\text{'find-element'}, 101)\}$$

Here each tuple contains a method name and a three-bit vector that is the semantic profile of the method. If we assume the tags of both 'field' and 'element' being 'type', rule set generation yields two interesting phrases: '`find`-*' and '`find`-[type]-*'. The phrases' corresponding rule sets follow from the semantic profiles: The '`find`-*' rule set enforces semantic property one and forbids semantic property two, but includes no rule for semantic property three since the corpus is inconclusive for methods matching '`find`-*'. However, in the narrower '`find`-[type]-*' sub-corpus property three is always present, and the '`find`-[type]-*' rule set will enforce it.

3. Lancelot

Our primary motivation for Lancelot has been to create a tool that lets Java programmers use our research results in day-to-day development work. We want to give them immediate feedback each time they write a method, as well as allow them to analyse methods in a batch run. We target Eclipse, a well-known and widely used open-source integrated development environment for Java programming.

We hope also that our experiences with making such a tool is of interest to other researches that want to make their program analyses available inside Eclipse.

3.1 Finding and fixing naming bugs

Lancelot finds and fixes naming bugs as a three-step process: Given some method we first analyse its method name using part-of-speech tagging, obtaining a phrase, and then we analyse the method implementation, yielding a semantic profile. This step is the same as when building the method corpus (Section 2.1). Then, in the second step, we search the rule book for a maximally specific matching phrase for the method's name. Last, in the third step, we consider each rule in the phrase's rule set, and record every case where the semantic profile disagrees with the omission or inclusion of a property mandated by the rule set. These cases are called rule *violations*.

While Lancelot's approach to *finding* naming bugs corresponds exactly to the one presented in our earlier work [4, 3], its approach to *fixing*[4] naming bugs differs slightly. In our previous work we relied upon data about all methods in the corpus when proposing naming bug fixes. This large amount of data is not practical to include in an Eclipse plugin. Instead, we construct an 'inverse rule book', that is, a partial function mapping a subset of semantic profiles to lists of 'suitable' phrases. We build this *semantic map* in two steps. First, we consider each phrase corpus (found as discussed in Section 2.2), and register all prevalent relations between semantic profiles and phrases. For instance, one such relation could be that the semantic profile $\langle \textit{Returns boolean}, \neg(\textit{Contains branch})\rangle$ occurs 43 times together with the name `is-consistent`. Second, we process this data set, building ordered 'suggestion lists': We sort the lists on prevalence, remove seldom-seen semantic profiles and reduce families of specific, but similar phrases to unifying, general forms. This pruning produces suggestion lists which mainly consist of the most prevalent basic verbs. Our experience is that, despite the pruning, this alternative approach does capture the essence of the previous corpus-based approach for fixing naming bugs.

[4] Rather than fixing naming bugs, Lancelot proposes suggestions as to how bugs may be fixed. It is up to the programmer to judge the suggestions and make the actual fix.

3.2 Lancelot's design

The Lancelot plugin consists of two main components: The *Lancelot Engine*, which implements the naming bug analysis as described in previous sections; and the *Lancelot Eclipse Plugin*, the component that integrates this engine into the Eclipse Java IDE. In addition to these two components, Lancelot also has an off-line component used for generating the rule book.

Lancelot Engine consists of several sub-systems:

- The analysis core, built around the ASM Bytecode framework.[5]
- Implementations of the *part-of-speech tagger*, *rule book* and *semantic map*, as specified in previous sections.
- The *facade*, the publicly exposed entry-point which orchestrates operation of the other sub-systems. Colloquially speaking, it is a component that produces naming bug reports given JVM bytecode.

Although currently packaged as an Eclipse plugin, Lancelot Engine is oblivious of the surrounding Eclipse context, and may easily be integrated in other contexts, for instance as an Ant or Maven plugin, or as a stand-alone tool.

We do not discuss the design of Lancelot's Eclipse-specific component, because such a discussion would require an introduction to Eclipse's plugin architecture, but we comment on our experience with targeting an interactive static analysis tool to Eclipse.

3.3 Notes on implementation

In general, the development of the Eclipse integration was unproblematic. Although a voluminous project, Eclipse is well-documented. However, we faced some difficulties. The primary one was managing the relation between Eclipse's various models of the code and Lancelot Engine's model: While extracting JVM bytecode from Eclipse's Java model proved easy, relating the types and methods in the analysis results to their corresponding elements in the model was definitely non-trivial (primarily because of type erasure). Although Eclipse has relatively good support for resolving type names in various contexts, we found that these procedures could be quite slow, and we had to implement alternative solutions in several places. A related and difficult problem is to identify anonymous classes. To address this, both Lancelot and FindBugs [5] use modifications of a 500-line snippet that originated in the Bytecode Outline plugin[6].

Despite the mentioned problems and the unfortunate workarounds, we are satisfied with the resulting design and implementation of the Eclipse integration. The great majority of the code could easily be re-used in other static analysis tools. Lancelot Eclipse Plugin specific parts currently comprises two thousand lines of Java code, while Lancelot Engine comprises five thousand.

4. Using Lancelot

Lancelot extends the context menus of Eclipse's Project/Package Explorer views with options for running analysis, clearing naming bug markers, and—if a project is selected—enabling or disabling whole-project automatic analysis.

[5] `http://asm.ow2.org/`

[6] `http://andrei.gmxhome.de/bytecode/`

Lancelot is closely integrated with Eclipse's Java development environment: Naming bugs show up in Java editors as *markers* on affected elements, while alternative name suggestions appear as *marker resolutions* or 'quick fixes'. Lancelot has two kinds of marker resolutions: First, to suppress false positive, there are 'suppress warnings' resolutions, which, when invoked, adds a Lancelot-specific `SuppressWarnings` annotation[7] to affected methods.[8] Second, there are 'renaming resolutions', which Lancelot adds after finding potentially better-fitting phrases during analysis. Invoking one of those opens Eclipse's standard 'Rename method' dialogue, so that the user may concretise and customise the proposed phrase.

Lancelot may run *automatically* or *manually*. In *automatic* mode, analysis runs each time a project is built, giving instant feedback during development. In *manual mode*, users manually select resources and invoke Lancelot through a menu option. Depending on task and intention, users will probably find one mode of operation more relevant than the other. Analysts and testers are likely to find manual mode the best fit, while developers actively working on code probably consider automatic feedback most useful.

4.1 Example use of Lancelot

To illustrate Lancelot, we analyse the sub-component of AspectJ 1.6.11 that contains the naming bug discussed in the introduction (specifically, this is the Apache Commons BCEL, included as part of AspectJ). In our explanation here, we focus on how the naming bug manifests itself in the analysis results, illustrating how the rule book violations and alternative name suggestions are presented.

First, we import the code for the BCEL component and its dependencies into Eclipse, and assert that Eclipse reports no build errors. Second, we select the package `org.aspectj.apache.bcel.generic` in the project menu and invoke Lancelot on the package from the context menu.

After about three seconds of processing, the package's naming bugs appear in Eclipse's problem marker listing.[9] Here Lancelot reports the naming bug discussed in the introduction, for the `containsField` method on the `ClassGen` class. If we then navigate to this bug's marker in the Java editor, Eclipse presents a pop-up dialogue with a textual description of the naming bug and several resolutions, as illustrated in Figure 2. First, if the user considers the bug report a false positive, she may insert Lancelot's `SuppressWarnings` annotation. Alternatively, the user may choose one of the alternative name suggestions and change the name of the method accordingly using Eclipse's standard renaming refactoring.

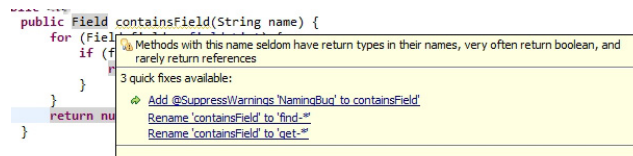

Figure 2. A naming bug with possible fixes for the method name.

[7] `http://download.oracle.com/javase/6/docs/api/java/lang/SuppressWarnings.html`

[8] Whether using in-code annotations for ignoring, as well as piggybacking onto `java.lang`'s `SuppressWarnings`, is a viable solution is open to debate. We decided to settle with this relatively simple solution, but we remain interested in alternative solutions.

[9] Alternatively, a very simple view that exclusively shows naming bug markers may be opened from the 'Show view' sub-menu in Eclipse's top bar.

5. Discussion

Further work on Lancelot will take two main directions: Improving rule book generation and improving the plugin itself. The rule book should be generated with a more recent and independent corpus, such as the Qualitas corpus [10]. Also, we would like to investigate the degree of 'convergence' of the rule book when enough applications are added to the corpus, similar to the central limit theorem of probability theory. Furthermore, we will investigate more closely the nature of naming bugs, for example, how can they be classified, and we will work on systematically measuring false positives.

We would also like to conduct a case study in which developers unfamiliar with Lancelot use the plugin on their own source code. This would allow us to learn more about user benefits and expectations, and it would point us to parts needing improvement in both Lancelot's user interface and the underlying theory.

References

[1] K. Beck. *Implementation Patterns*. Addison-Wesley Professional, 2007.

[2] C. Fellbaum. *WordNet: An Electronic Lexical Database*. MIT Press, 1998.

[3] E. W. Høst. *Meaningful Method Names*. PhD thesis, University of Oslo, 2011. `http://urn.nb.no/URN:NBN:no-27629`.

[4] E. W. Høst and B. M. Østvold. Debugging method names. In S. Drossopoulou, editor, *Proceedings of the 23rd European Conference on Object-Oriented Programming*, volume 5653 of *Lecture Notes in Computer Science*, pages 294–317. Springer, 2009.

[5] D. Hovemeyer and W. Pugh. Finding bugs is easy. In J. M. Vlissides and D. C. Schmidt, editors, *OOPSLA Companion*, pages 132–136. ACM, 2004.

[6] D. Lawrie, C. Morrell, H. Feild, and D. Binkley. What's in a name? A study of identifiers. In *Proceedings of the 14th International Conference on Program Comprehension (ICPC 2006), 14-16 June 2006, Athens, Greece*, pages 3–12. IEEE Computer Society, 2006.

[7] C. D. Manning and H. Schuetze. *Foundations of Statistical Natural Language Processing*. MIT Press, 1999.

[8] R. C. Martin. *Clean Code*. Prentice Hall, 2008.

[9] S. McConnell. *Code Complete: A Practical Handbook of Software Construction*. Microsoft Press, 2nd edition, 2004.

[10] E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble. Qualitas corpus: A curated collection of Java code for empirical studies. In *2010 Asia Pacific Software Engineering Conference (APSEC2010)*, Dec. 2010.

A. Getting Lancelot

Lancelot is available under the terms of the Eclipse Public License in both compiled and source form. Stable binaries reside in our Eclipse update site[10], while source code is hosted in a publicly available Subversion repository at the Lancelot website[11].

Lancelot targets Eclipse's 2011 annual release, Indigo, and runs in all environments supporting Eclipse. Lancelot manages all its dependencies, requiring only a working Eclipse installation with the standard Java Developer Tools for installation and execution.

Although the Eclipse update site is likely the best option for the majority of users, interested developers could alternatively check out Lancelot's code from the source repository and into Eclipse, and run the tool directly in Eclipse's Plug-in Development Environment.

[10] `http://lancelot.nr.no/`

[11] `http://code.google.com/p/lancelot-eclipse/`