Debugging Method Names

Einar W. Høst and Bjarte M. Østvold

Norwegian Computing Center
{einarwh,bjarte}@nr.no

Abstract. Meaningful method names are crucial for the readability and maintainability of software. Existing naming conventions focus on syntactic details, leaving programmers with little or no support in assuring meaningful names. In this paper, we show that naming conventions can go much further: we can mechanically check whether or not a method name and implementation are likely to be good matches for each other. The vast amount of software written in Java defines an implicit convention for pairing names and implementations. We exploit this to extract rules for method names, which are used to identify "naming bugs" in well-known Java applications. We also present an approach for automatic suggestion of more suitable names in the presence of mismatch between name and implementation.

1 Introduction

It is well-known that maintenance costs dominate — if not the budget — then the true cost of software [7]. It is also known that code readability is a vital factor for maintenance [5]: unintelligible software is necessarily hard to modify and extend. Finally, it has been demonstrated that the quality of identifiers has a profound effect on program comprehension [14]. We conclude that identifier quality affects the cost of software! Hence, we would expect programmers to have powerful analyses and tools available to help assure that identifier quality is high.

The reality is quite different. While the importance of good names is undisputed among leading voices in the industry [2, 18, 19], the analyses and tools are lacking. Programmer guidance is limited to naming convention documents such as those provided by Sun Microsystems for Java. The following quote is typical for the kind of advice given by such documents: "Except for variables, all instance, class, and class constants are in mixed case with a lowercase first letter"[1]. In other words, the documents mandate a certain uniformity of lexical syntax. Since such uniformity is easily checked mechanically, there are tools available to check for violations against these rules. While this is certainly useful, it does little to ensure meaningful identifiers. (Arguably, syntactic uniformity helps reduce the cost of "human parsing" of identifiers, but not the interpretation.) Since identifiers clearly must be meaningful to be of high quality, current tool-support must be considered unsatisfactory.

[1] http://java.sun.com/docs/codeconv/html/CodeConventions.doc8.html

This begs the question what meaningful identifiers really are. Consider what an identifier is used for: it represents some program entity, and allows us to refer to that entity by means of the identifier alone. In other words, the identifier is an abstraction, and the meaning relates to the program entity it represents. The identifier is meaningful if the programmer can interpret it to gain an understanding of the program entity without looking at the entity itself. Intuitively, we also demand that the abstraction be sound: we must agree that the identifier is a suitable replacement for the entity. Hence, what we really require are identifiers that are both *meaningful* and *appropriate*.

In this work, we consider only method names. Methods are the smallest named units of aggregated behaviour in most conventional programming languages, and hence a cornerstone of abstraction. A method name is meaningful and appropriate if it adequately describes the implementation of the method. Naming is non-trivial because there is a potential for conflict between names and implementations: we might choose an inappropriate name for an implementation, or provide an inappropriate implementation for a name. The label *appropriate* is not really a binary decision: there is a sliding scale from the highly appropriate to the utterly inappropriate. Inappropriate or even meaningless identifiers are obviously bad, but subtle mistakes in naming can be as confusing or worse. Since the programmer is less likely to note the subtle mistake, a misconception of the code's behaviour can be carried for a long time.

Consider the following example, taken from AspectJ 1.5.3, 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;
 }
```

Most Java programmers will find it easy to come up with a name for this method: clearly, this is a *find* method! More precisely, we would probably name this method `findField`; a suitable description for a method that indeed tries to find a `Field`. The name used in AspectJ, however, is `containsField`. We consider this to be a *naming bug*, since the name indicates a question to the object warranting a boolean reply ("Do you contain a field with this name?") rather than an instruction to return an object ("Find me the field with this name!"). In this paper, we show how to derive rules for implementations of *contains* methods, *find* methods and other methods with common names, allowing us to identify this naming bug and many others. We also present an approach for

automatic correction of faulty names that successfully suggests using the verb *find* rather than *contains* for the code above.

It is useful to speak of method names in slightly abstract terms; for instance, we speak of *find* methods, encompassing concrete method names like `findField` and `findElementByID`. We have previously introduced the term *method phrase* for this perspective [12]. Typically, the rules uncovered by our analysis will refer to method phrases rather than concrete method names. This is because method phrases allow us to focus on essential similarities between method names, while ignoring arbitrary differences.

The main contributions of this paper are as follows:

- A formal definition of a naming bug (Sect. 3.1).
- An approach for encoding the semantics of methods (Sect. 3.3), building on our previous work [12, 11].
- An approach for extracting name-specific implementation rules for methods (Sect. 3.4).
- An automatically generated "rule book" containing implementation rules for the most common method names used in Java programming (Sect. 3.4).
- An approach for automatic suggestion of a more suitable name in the case of mismatch between the name and implementation of a method (Sect. 3.6).

We demonstrate the usefulness of our analysis by finding genuine naming bugs in well-known Java applications (Sect. 5.2).

2 Motivation

Our goal is to exploit the vast amount of software written in Java to derive name-specific implementation rules for methods. Our approach is to compare the names and implementations of methods in a large corpus of well-known open-source Java applications. In this section, we motivate our approach, based on philosophical considerations about the meaning of natural language expressions.

2.1 The Java Language Game

We have previously argued that method identifiers act as hosts for expressions in a natural language we named *Programmer English* [12]. Inspired by Wittgenstein and Frege, we take a pragmatic view of how meaning is constructed in natural language. According to Wittgenstein, "the meaning of a word is its use in the language" [27]. In other words, the meaning is simply the sum of all the uses we find of the word — there is no "objective" definition apart from this sum. It follows that meaning is not static, since new examples of use will skew the meaning in their own direction. Also, any attempt at providing a definition for a word (for instance in a dictionary, or our own phrase book for Java [12]) is necessarily an imperfect approximation of the meaning.

Wittgenstein used the term *language game* (Sprachspiel) to designate simple forms of language, "consisting of language and the actions into which it is

woven" [27]. Intuitively, a language game should be understood as interplay between natural language expressions and behaviours. Hence, our object of inquiry is really the Java language game, where the language expressions are encoded in method identifiers and the actions are encoded in method bodies.

In discussing the meaning of symbolic language expressions, Frege [9] introduces the terms *sign*, *reference* and *sense*. The sign is the name itself, or a combination of words. The reference is the object to which the sign refers. The sense is our collective understanding of the reference. In the context of Java programming, we take the sign to be the method phrase, the reference to be the "true meaning" indicated by that phrase (that Wittgenstein would claim is illusory), and the sense to be the Java community's collective understanding of what the phrase means. Of course, the collective understanding is really unavailable to us: we are left with our own subjective and imperfect understanding of the sign. This is what Frege refers to as the individual's *idea*. Depending on our level of insight, that idea may be in various degrees of harmony or conflict with the actual sense.

Interestingly, when analysing Java methods, we *do* have direct access to a manifestation of the programmer's idea of the method name's sense: the method body. By collecting and analysing a large number of such ideas, we can approximate the sense of the name. This, in turn, allows us to identify naming bugs: ideas that are in conflict with the approximated sense.

3 Analysis of Methods

We turn our understanding of how meaning is constructed into a practical approach for approximating the meaning of method names in Java. This approximation is then used to create rules for method implementations. Finally, these rules help us identify naming bugs. Fig. 1 provides an overview of the approach. The analysis consists of three major phases: data preparation, mining of implementation rules, and identification of naming bugs.

In the data preparation phase, we transform our corpus of Java applications into an idealised corpus of methods. The transformation entails analysing each Java method in two ways. On the one hand, we perform a natural language analysis on the method name (Sect. 3.2). This involves decomposing the name into individual words and performing part-of-speech tagging of those words. The tags allow us to form abstract phrases from the concrete method names. On the other hand, we analyse the signature and Java bytecode of the method implementation, deriving a semantic profile for each implementation (Sect. 3.3).

This sets us up to investigate the semantics of methods that share the same abstract phrase. We start with very abstract phrases that we gradually refine into more concrete phrases, more closely matching the actual method names. If a given phrase fulfils certain criteria pertaining to prevalence, we derive a corresponding set of implementation rules (Sect. 3.4) that all methods whose names match the phrase must obey. Failure to obey an implementation rule is considered a naming bug (Sects. 3.5 and 3.6).

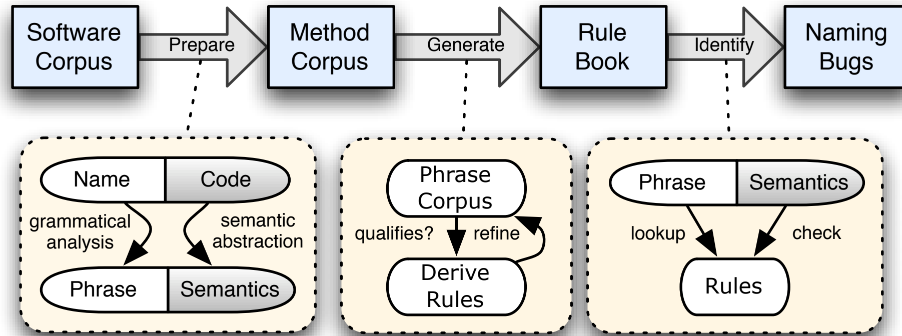


Fig. 1: Overview of the approach.

3.1 Definitions

In the following, please refer to Fig. 2 for an overview of the relationships between the introduced terms.

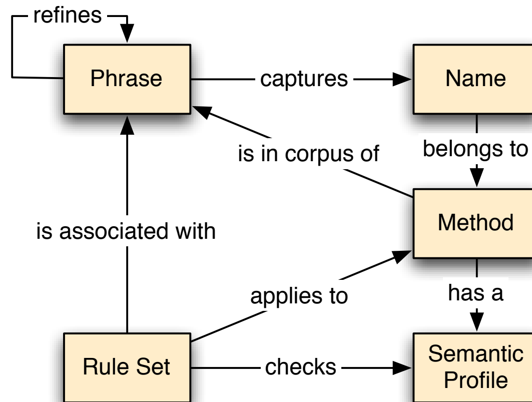


Fig. 2: Conceptual model of phrase terms.

We define a *method* m as a tuple consisting of a unique *fingerprint* u, a *name* n, and a *semantic profile* $[\![m]\!]$. The unique fingerprints prevent set elements from collapsing into one; hence, a set made from arbitrary methods $m_1, \ldots, m_k$ will always have k elements. The name n is a non-empty list of *fragments* f. Each fragment is annotated with a *tag* t.

The semantic profile $[\![m]\!]$ for a method m is defined in terms of *attributes*. We define a set $\mathcal{A}$ of attributes $\{a_1, \ldots, a_k\}$, and let a denote an attribute from $\mathcal{A}$. Given a method m and an attribute a, the expression $check(m, a)$ is a binary value $b \in \{0, 1\}$. Intuitively, *check* determines whether or not m fulfils the predicate defined by a. We then define $[\![m]\!]$ as the list $[check(m, a_1), \ldots, check(m, a_k)]$. It follows that there are at most $2^{|\mathcal{A}|}$ distinct semantic profiles. The *rank* of a semantic profile in a corpus is the proportion of methods that have that semantic profile.

A *phrase* p is a non-empty list of *parts* ρ; its purpose is to abstract over method names. A part ρ may be a fragment f, a tag t, or a special wildcard symbol $*$. The wildcard symbol may only appear as the last part of a phrase. A phrase that consists solely of fragments is *concrete*; all other phrases are *abstract*.

A phrase *captures* a name if each individual part of the phrase captures each fragment of the name, in order from first to last. A fragment part captures a fragment if they are equal. A tag part captures a fragment if it is equal to the fragment's tag. A wildcard part captures any remaining fragments in a name, including zero fragments. A concrete phrase can only capture a single name, whereas an abstract phrase can capture multiple names. For instance, the abstract phrase **is-⟨adjective⟩-*** captures names like **is-empty**, **is-valid-signature** and so forth.

A *corpus* $\mathcal{C}$ is a set of methods. Implicitly, $\mathcal{C}$ defines a set $\mathcal{N}$, consisting of the names of the methods $m \in \mathcal{C}$. A *name corpus* $\mathcal{C}_n$ is the subset of $\mathcal{C}$ with the name n. Similarly, a *phrase corpus* $\mathcal{C}_p$ is the subset of $\mathcal{C}$ whose names are captured by the phrase p. The *frequency value* $\xi_a(\mathcal{C})$ for an attribute a given a corpus $\mathcal{C}$ is defined as:

$$\xi_a(\mathcal{C}) \stackrel{\text{def}}{=} \frac{\sum_{m \in \mathcal{C}} check(m, a)}{|\mathcal{C}|}$$

The semantics of a corpus $\mathcal{C}$ is defined as the list $[\xi_{a_1}(\mathcal{C}), \ldots, \xi_{a_k}(\mathcal{C})]$. We write $[\![p]\!]_{\mathcal{C}}$ for the semantics of a phrase in corpus $\mathcal{C}$, and define it as the semantics of the corresponding phrase corpus. The subscript will be omitted when there can be no confusion as to which corpus we refer to.

We introduce a subset $\mathcal{A}_o \subset \mathcal{A}$ of *orthogonal attributes*. Two attributes a_1 and a_2 are considered orthogonal if $check(m, a_1)$ does not determine $check(m, a_2)$ or vice versa for any method m. We define the *semantic distance* $d(p_1, p_2)$ between two phrases p_1 and p_2 as the vector distance

$$d(p_1, p_2) \stackrel{\text{def}}{=} \sum_{a \in \mathcal{A}_o} \left(\xi_a(\mathcal{C}_{p_1}) - \xi_a(\mathcal{C}_{p_2})\right)^2$$

A *rule* r is a tuple consisting of an attribute a, a *trigger condition* c and a *severity* s. The trigger condition c is a binary value, indicating whether the rule is triggered when the function *check* evaluates to 0 or to 1. The severity s is defined as $s \in \{forbidden, inappropriate, reconsider\}$. For example, the rule $r = (a_{reads_field}, 1, inappropriate)$ indicates that it is considered *inappropriate* for the **reads field** attribute to evaluate to 1. Applied to a method implementation,

the rule states that the implementation should not read field values. In practice, rules are relevant for specific phrases. Hence, we associate with each phrase p a set of rules $\mathcal{R}_p$ that apply to the methods $m \in \mathcal{C}_p$.

Finally, we define a boolean function $bug(r, m) \stackrel{\text{def}}{=} check(m, a) = c$ that evaluates to true when the rule $r = (a, c, s)$ is triggered by method m.

3.2 Analysing Method Names

Far from being arbitrary labels, method names act as hosts for meaningful phrases. This is the premise we rely on when we state that it is possible to define name-specific rules for the implementation of methods. According to Liblit [15], "[method] names exhibit regularities derived from the grammars of natural languages, allowing them to combine together to form larger pseudo-grammatical phrases that convey additional meaning about the code". To reconstruct these phrases, we decompose the method names into individual fragments, and apply a natural language processing technique called part-of-speech tagging [17] to identify their grammatical structure.

Decomposition. By convention, Java programmers use "camel case" when forming method names that consist of multiple fragments ("words"). A camel case method name uses capitalised fragments to compensate for the lack of whitespace in identifiers. For instance, instead of writing `create new instance` (which would be illegal), Java programmers write `createNewInstance`. To recover the individual fragments, we reverse the process, using capital characters as an indicator to split the name, with special treatment of uppercase acronyms. For instance, we decompose `parseXMLNode` into `parse XML node` as one would expect. Some programmers use underscore as delimiter instead of case-switching; however, we have previously noted that this is quite rare [12]. For simplicity, we therefore choose to omit such methods from the analysis.

Part-of-speech Tagging. Informally, part-of-speech tagging refers to the process of tagging each word in a natural language expression with information about its the grammatical role in the expression. In our scenario, this translates to tagging each fragment in the decomposed method name. We consider a decomposed method name to be an untagged method phrase.

An overview of the tagging process is shown in Fig. 3. First, we use the tags **verb**, **noun**, **adjective**, **adverb**, **pronoun**, **preposition**, **conjunction**, **article**, **number**, **type** and **unknown** to tag each fragment in the phrase. In other words, apart from the special tags **number**, **type** and **unknown**, we use the basic word classes. The **number** tag is used for numeric fragments like **1**. The **type** tag is used when we identify a fragment as the name of a type in scope of the method. Fragments that we fail to tag default to the **unknown** tag.

We make three attempts at finding suitable tags for a fragment. First, we use WordNet [8], a large lexical database of English, to find verbs, nouns, adjectives

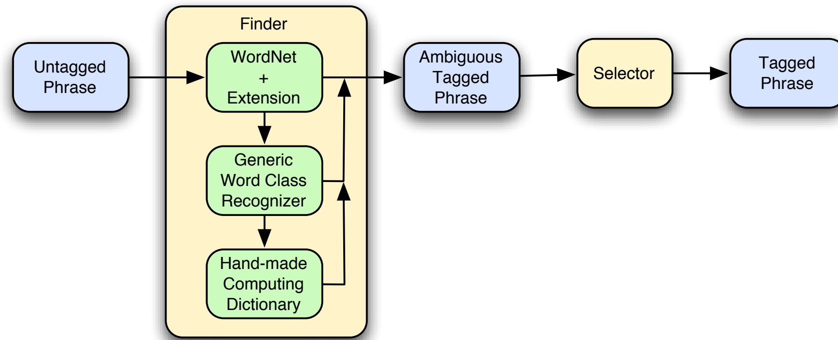


Fig. 3: Part-of-speech tagging for method phrases.

and adverbs. We augment the results given by WordNet with lists of pronouns, prepositions, conjunctions and articles. If we fail to find any tags, we use a mechanism for identifying invented words. Programmers sometimes derive nouns and adjectives from verbs (for instance, **handler** from **handle** and **foldable** from **fold**), or verbs from nouns (for instance, **tokenize** from **token**). If we can discover such derivations, we tag the fragment accordingly. Finally, we resort to a manual list of tags for commonly used programming terms.

Since a fragment may receive multiple tags (for instance, WordNet considers **object** to be both a noun and a verb), the initial tagging leads to an ambiguously tagged phrase. We then perform a selection of tags that takes into account both the fragment's position in the phrase, and the tags of surrounding fragments. This yields an unambiguously tagged phrase. We have previously estimated the accuracy of the part-of-speech tagger to be approximately 97% [12].

Method Phrases and Refinement. The decomposed, tagged method names are concrete method phrases. The tags allow us to form abstract phrases as well; phrases where concrete fragments have been replaced by tags. Phrases are written like this: **get-⟨noun⟩-***, where the individual parts are separated by hyphens, fragments are written straightforwardly: **get**, tags are written in angle brackets: ⟨**noun**⟩, and the * symbol indicates that the phrase can be further refined.

Refinement involves reducing the corresponding phrase corpus to a subset. In general, there are three kinds of refinement:

1. Introduce tag: **p-*** ⇒ **p-⟨t⟩-***.
   For instance, the phrase **is-*** may be refined to **is-⟨adjective⟩-***. The former phrase would capture a name like `isObject`, the latter would not.
2. Remove wildcard: **p-*** ⇒ **p**.
   For instance, the phrase **is-⟨adjective⟩-*** may be refined to **is-⟨adjective⟩**.

The former phrase would capture a name like `isValidSignature`, the latter would not.

3. Replace tag with fragment: **p-⟨t⟩-*** ⇒ **p-f-***.
   For instance, the phrase **is-⟨adjective⟩-*** may be refined to **is-empty-***. The former phrase would capture a name like `isValid`, the latter would not.

Fig. 4 shows the refinement steps leading from the completely abstract phrase *, to the concrete phrase **is-empty**. When we reach a concrete phrase, we attempt a final step of further refinement to annotate the concrete phrase with information about the types of return value and parameters. Hence we can form signature-like phrases like **boolean is-empty()**. This step is not included in the figure, nor in the list above.

Fig. 4: The refinements leading to **is-empty**.

3.3 Analysing Method Semantics

In any data mining task, the outcome of the analysis depends on the domain knowledge of the analyst [26]. Hence, we must rely on our knowledge of Java programming when modelling the semantics of methods. In particular, we consider some aspects of the implementation to be important clues as to the behaviour of methods, whereas others are considered insignificant.

A method m has some basic behaviours pertaining to data flow and control flow that we would like to capture: 1) read or write fields, 2) create new objects, 3) return a value to the caller, 4) call methods, 5) branch and/or repeat iteration, and 6) catch and/or throw exceptions. We concretise the basic behaviours by means of a list of machine-traceable *attributes*, formally defined as predicates on Java bytecode. In addition to the attributes stemming from the basic behaviours, called *instruction attributes*, we define a list of *signature attributes*. Table 1 lists all the attributes, coarsely sorted in groups. Note that some attributes, such as **returns created object** really belong to more than one group. Attributes marked with an asterisk belong to the subset of orthogonal attributes.

Most of the attributes should be fairly self-explanatory; however, the attributes pertaining to object creation warrant further explanation. A regular object is an object that does not inherit from the type `java.lang.Throwable`, a string object is an instance of the type `java.lang.String`, and a custom object is one that does not belong to either of the namespaces `java.*` and `javax.*`. Finally, the attribute **creates own type objects** indicates that the method creates an instance of the class on which the method is defined.

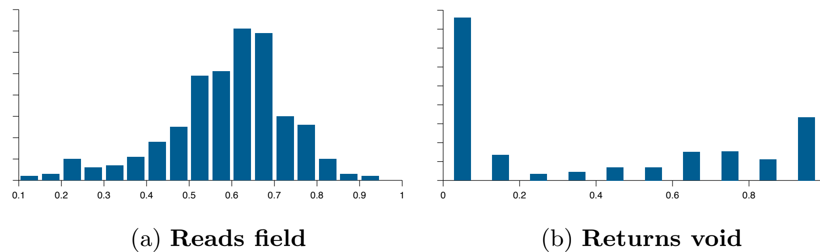
Table 1: Attributes. Orthogonal attributes marked with an asterisk.

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

3.4 Deriving Phrase-Specific Implementation Rules

We derive a set of implementation rules for method phrases that are *prevalent* in a large corpus of Java applications. A phrase is considered prevalent if it fulfils a simple heuristic: it must occur in at least half of the applications in the corpus, and it must cover at least 100 method instances. While somewhat arbitrary, this heuristic guards against idiosyncratic naming in any single application, and ensures a fairly broad basis for the semantics of the phrase. Each prevalent phrase is included in a conceptual "rule book" derived from the corpus, along with a corresponding set of rules. Intuitively, all methods captured by a certain phrase must obey its implementation rules.

We define the implementation rules on the level of individual attributes. To do so, we consider the frequency values of the attributes for different phrase corpora. The intuition is that for a given phrase corpus, the frequency value for an attribute indicates the probability for the attribute's predicate to be fulfilled for methods in that corpus. For each attribute $a \in \mathcal{A}$, we find that the the frequency value $\xi_a(\mathcal{C}_n)$ is distributed within the boundaries $0 \leq \xi_a(\mathcal{C}_n) \leq 1$. We assume that method names therefore can be used to predict whether or not an attribute will evaluate to 1: different names lead to different frequency values. Fig. 5 shows example distributions for the attributes **reads field** and **returns void** for some corpus. We see that the two distributions are quite different. Both attributes distinguish between names, but **returns void** is clearly the most polarising of the two for the corpus in question.



(a) **Reads field** (b) **Returns void**

Fig. 5: Distribution of frequency values for two attributes.

A frequency value close to 0 indicates that it is rare for methods in the corresponding corpus to fulfil the predicate defined by the attribute; a value close to 1 indicates the opposite. We exploit this to define rules. Any method that deviates from the norm set by the phrase corpus to which it belongs is suspect. If the norm is polarised (close to 0 or 1), we induce a rule stating that the attribute should indeed evaluate to only the most common value. Breaking a rule constitutes a naming bug. Note that there are two kinds of naming bugs, that we call *inclusion bugs* and *omission bugs*. The former refers to methods that

fulfil the predicate of an attribute it should not, the latter to methods that fail to fulfil a predicate it should. We expect inclusion bugs to be more common (and arguably more severe) than omission bugs. For instance, it might be reasonable to refrain from doing anything at all (an empty method) regardless of name, whereas throwing an exception from a seemingly innocent `hasChildren` method is more dubious.

Specifically, we induce rules by defining percentiles on the distribution of frequency values for each attribute $a \in \mathcal{A}$. The percentiles are 0.0%, 2.5%, 5.0%, 95.0%, 97.5% and 100.0%, and are associated to a degree of severity when the corresponding rules are violated (see Table 3.4). The intuition is that the percentiles classify the frequency values of different phrases relative to each other. Assume, for instance, that we have a corpus $\mathcal{C}$ and a phrase p with a corresponding corpus $\mathcal{C}_p \subset \mathcal{C}$ of methods yielding a frequency value $\xi_a(\mathcal{C}_p)$ for a certain attribute $a \in \mathcal{A}$. Now assume that the frequency value belongs to the lower 2.5% when compared to that of other phrases in $\mathcal{C}$. Then we deem it *inappropriate* for a method $m \in \mathcal{C}_p$ to fulfil the predicate defined by a.

Table 2: Percentile groups for frequency values.

Percentiles (%)	*Severity*
0.0	Forbidden (if included)
0.0 − 2.5	Inappropriate (if included)
2.5 − 5.0	Reconsider (if included)
5.0 − 95.0	No violation
95.0 − 97.5	Reconsider (if omitted)
97.5 − 100.0	Inappropriate (if omitted)
100.0	Forbidden (if omitted)

3.5 Finding Naming Bugs

Once a set of rules has been obtained for each prevalent phrase in the corpus, finding naming bugs is trivial. For each of the methods we want to check, we attempt to find the rule set for the most concrete capturing phrase (see Fig. 2). In a few cases, the capturing phrase may be fully concrete, so that it perfectly matches the method name. This is likely to be the case for certain ubiquitous method names and signatures such as `String toString()` and `int size()`, for instance. In most other cases, we expect the phrase to be more abstract. For instance, for the method name `Element findElement()`, the most concrete capturing phrase might be something like **ref find-⟨type⟩**. Failure to find any capturing phrase at all could be considered a special kind of naming bug; that the name itself is rather odd.

When we have found the most concrete capturing phrase p, we obtain the corresponding rule set $\mathcal{R}_p$ that applies to the method. For each rule in the rule

set, we pass the rule and the method to the function *bug*. Whenever *bug* returns true, we have a rule violation, and hence a naming bug. Note that a single method might violate several implementation rules, yielding multiple naming bugs.

3.6 Fixing Naming Bugs

Naming bugs manifest themselves as violations of phrase-specific implementation rules. A rule violation indicates a conflict between the name and the implementation of a method. There are two ways to resolve the conflict: either we assume that the name is correct and the implementation is broken, or vice versa. The former must be fixed by removing offending or adding missing behaviour. While it is certainly possible to attempt to automate this procedure, it is likely to yield unsatisfactory or even wrong results. The programmer should therefore attend to this manually, based on warnings from the analysis.

We are more likely to succeed, at least partially, in automating the latter. We propose the following approach to find a suitable replacement name for an implementation that is assumed to be correct. The implementation is represented by a certain semantic profile. Every prevalent phrase that has been used for that profile is considered a *relevant* phrase for replacement. Some of the relevant phrases may be unsuitable, however, because they have rules that are in conflict with the semantic profile. We therefore filter the relevant phrases for rule violations against the semantic profile. The resulting list of phrases are *candidates* for replacement. Note that, in some cases, the list may be empty. If so, we deem the semantic profile to be *unnameable*.

Finding the best candidate for replacement is a matter of sorting the candidate list according to some criterion. We consider three relevant factors: 1) the rank of the semantic profile in the candidate phrase corpus, 2) the semantic distance from the inappropriate phrase to the candidate phrase, and 3) the number of syntactic changes we must apply to the inappropriate phrase to reach the candidate phrase. We assume that the optimal sorting function would take all three factors — and possibly others — into consideration. As a first approximation to solving the problem, however, we suggest simply sorting the list according to profile rank and semantic distances separately, and letting the programmer choose the most appropriate of the two.

4 The Corpus

The main requirements for the corpus are as follows:

- It must be representative of real-world Java programming.
- It must cover a variety of applications and domains.
- It must include most well-known and influential applications.
- It must be large enough to be credible as establishing “canonical” use of method names.

Table 3 lists the 100 Java applications, frameworks and libraries that constitute our corpus. Building and cleaning a large corpus is time-consuming labour; hence we use the same corpus that we have used in our previous work [12, 11]. The corpus was constructed to cover a wide range of application domains and has been carefully pruned for duplicate code. The only alteration we have made in retrospect is to remove a large number of near-identical code-generated `parse` methods from XBeans and Geronimo. The code clones resulted in visibly skewed results for the **parse-*** phrase, and proves that code generation is a real problem for corpus-based data mining.

Some basic numbers about the pruned corpus are listed in Table 4. We omit methods flagged as synthetic (generated by the compiler) as well as methods with "non-standard names". We consider a standard name be at least two characters long, start with a lowercase letter, and not contain any dollar signs or underscores.

5 Results

Here we present results from applying the extracted implementation rules on the corpus, as well as a small set of additional Java applications. In general, the rules can be applied to any Java application or library. For reasons of practicality and scale, however, we focus primarily on bugs in the corpus itself. We explain in detail how the analysis automatically identifies and reports a naming bug, and proceeds to suggest a replacement phrase to use for the method. We then investigate four rather different kinds of naming bugs revealed by the analysis. Finally, we present some overall naming bug statistics, and discuss the validity of the results.

5.1 Name Debugging in Practice

We revisit the example method from the introduction, and explain how the analysis helps us debug it.

```
public Field containsField(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;
}
```

Recall that we manually identified this as a *naming bug*, since we expect **contains-*** methods to return boolean values. Intuition tells us that *find* would be a more appropriate verb to use.

Table 3: The corpus of Java applications and libraries.

Desktop applications			
ArgoUML 0.24	Azureus 2.5.0	BlueJ 2.1.3	Eclipse 3.2.1
JEdit 4.3	LimeWire 4.12.11	NetBeans 5.5	Poseidon CE 5.0.1
Programmer tools			
Ant 1.7.0	Cactus 1.7.2	Checkstyle 4.3	Cobertura 1.8
CruiseControl 2.6	Emma 2.0.5312	FitNesse	JUnit 4.2
Javassist 3.4	Maven 2.0.4	Velocity 1.4	
Languages and language tools			
ANTLR 2.7.6	ASM 2.2.3	AspectJ 1.5.3	BSF 2.4.0
BeanShell 2.0b	Groovy 1.0	JRuby 0.9.2	JavaCC 4.0
Jython 2.2b1	Kawa 1.9.1	MJC 1.3.2	Polyglot 2.1.0
Rhino 1.6r5			
Middleware, frameworks and toolkits			
AXIS 1.4	Avalon 4.1.5	Google Web Toolkit 1.3.3	JXTA 2.4.1
JacORB 2.3.0	Java 5 EE SDK	Java 6 SDK	Jini 2.1
Mule 1.3.3	OpenJMS 0.7.7a	PicoContainer 1.3	Spring 2.0.2
Sun WTK 2.5	Struts 2.0.1	Tapestry 4.0.2	WSDL4J 1.6.2
Servers and databases			
DB Derby 10.2.2.0	Geronimo 1.1.1	HSQLDB	JBoss 4.0.5
JOnAS 4.8.4	James 2.3.0	Jetty 6.1.1	Tomcat 6.0.7b
XML tools			
Castor 1.1	Dom4J 1.6.1	JDOM 1.0	Piccolo 1.04
Saxon 8.8	XBean 2.0.0	XOM 1.1	XPP 1.1.3.4
XStream 1.2.1	Xalan-J 2.7.0	Xerces-J 2.9.0	
Utilities and libraries			
Batik 1.6	BluePrints UI 1.4	c3p0 0.9.1	CGLib 2.1.03
Ganymed ssh b209	Genericra	HOWL 1.0.2	Hibernate 3.2.1
JGroups 2.2.8	JarJar Links 0.7	Log4J 1.2.14	MOF
MX4J 3.0.2	OGNL 2.6.9	OpenSAML 1.0.1	Shale Remoting
TranQL 1.3	Trove	XML Security 1.3.0	
Jakarta commons utilities			
Codec 1.3	Collections 3.2	DBCP 1.2.1	Digester 1.8
Discovery 0.4	EL 1.0	FileUpload 1.2	HttpClient 3.0.1
IO 1.3.1	Lang 2.3	Modeler 2.0	Net 1.4.1
Pool 1.3	Validator 1.3.1		

Table 4: Basic numbers about the corpus.

JAR files	1003
Class files	189941
Candidate methods	1226611
Included methods	1090982

Finding the Bug. The analysis successfully identifies this as a naming bug, in the following way. First, we analyse the method. The name is decomposed into the fragments "contains" and "Field", which are tagged as **verb** and **type**, respectively. From the implementation, we extract a semantic profile that has the following attributes from Table 1 evaluated to 1, denoting presence: **return type in name**, **reads field**, **runtime type-check**, **contains loop**, **has branches**, **multiple returns**, **method call on field**. The rest of the attributes are evaluated to 0, denoting absence. We see that the attributes conspire to form an abstract description of the salient features of the implementation.

Table 5: Rules for **contains-*** methods.

Attribute	*Condition*	*Severity*	*Violation*
Returns void	1	Forbidden	
Returns boolean	0	Inappropriate	Yes
Returns string	1	Inappropriate	
Returns reference	1	Reconsider	Yes
Return type in name	1	Inappropriate	Yes
Parameter type in name	1	Reconsider	
Writes field	1	Reconsider	
Returns created object	1	Forbidden	
Creates own class objects	1	Inappropriate	

The most suitable phrase in our automatically generated rule book corresponding to the concrete phrase **contains-Field** is the abstract phrase **contains-***. The rule set for **contains-*** is listed in Table 5, along with the violations for the semantic profile. The mismatch between the name and implementation in this case manifests itself as three naming bugs. A **contains-*** should not return a reference type (much less echo the name of that type in the name of the method); rather, it should return a boolean value.

Fixing the Bug. There are two ways to fix a naming bug; either by changing the implementation, i.e., by returning a boolean value if the `Field` is found (rather than the `Field` itself), or by changing the name. In Sect. 3.6 we describe the approach for automatic suggestion of bug-free method names, to assist in the latter scenario.

Consider the top ten candidate replacement phrases listed in Table 6. An immediate reaction is that the candidates are fairly similar, and that *all* of them seem more appropriate than the original. Here we have sorted the list according to the sum of the orders given by the two ordering metrics *semantic distance* and *profile rank*; in cases of equal sum, we have arbitrarily given precedence to the phrase with the highest rank. In this particular example, we see that a rank ordering gives the better result, by choosing **ref find-⟨type⟩** over the more generic **find-⟨noun⟩-***.

Table 6: Candidate replacement phrases.

Phrase	*Distance*	*Rank*	*Sum*
find-⟨type⟩	4	3	7
find-*	2	5	7
ref find-⟨type⟩	7	1	8
find-⟨type⟩-*	5	4	9
find-⟨adjective⟩-*	3	6	9
ref find-⟨type⟩-*	8	2	10
find-⟨noun⟩-*	1	9	10
get-⟨type⟩-*(String...)	6	8	14
ref get-⟨type⟩-*(String...)	9	7	16
ref get-⟨type⟩-*	10	10	20

5.2 Notable Naming Bugs

To illustrate the diversity of naming bugs the phrase-specific implementation rules help us find, we explore a few additional examples of naming bugs found in the corpus. The four methods shown in Fig. 6 exhibit rather different naming bugs. Note that since both strategies for choosing replacement phrases yield similar results, we have included only the top candidate according to profile rank in the figure.

The first example, taken from Ant 1.7.0, is representative of a fairly common naming bug: the inappropriately named "boolean setter". While both Java convention and the JavaBean specification[2] indicate that the verb *set* should be used for all methods for writing properties (including boolean ones), programmers sometimes use an inappropriate **is-*** form instead. This mirrors convention in some other languages such as Objective-C, but yields the wrong expectation when programming Java. The problem is, of course, that `isCaching` reads like a question: "is it true that you are caching?". We expect the question to be answered. The analysis indicates three rule violations for the method, and suggests using the phrase **set-⟨adjective⟩-*** instead.

The second example, taken from the class `Value` in JXTA 2.4.1, shows a broken implementation of an `equals` method. According to Sun's documentation, "The equals method implements an equivalence relation on non-null object references"[3]: it should be reflexive, symmetric, transitive and consistent. It turns out that this is notoriously hard to implement correctly. An influential book on Java devotes much attention to the details of fulfilling this contract [3]. The problem with the implementation from JXTA is that it is *not* symmetric, and the symptom is the creation of an instance of the type that defines the method. Assume that we have a `Value` instance v. The last instruction returns true whenever the parameter can be serialised to a String that in turn is used to create a `Value`

[2] `http://java.sun.com/javase/technologies/desktop/javabeans/docs/spec.html`

[3] `http://java.sun.com/j2se/1.4.2/docs/api/java/lang/Object.html`

// Ant 1.7.0
public void isCaching(boolean value) {
this.caching = value;
}
	Semantic Profile:
	Returns void, Writes field, Parameter to field
	Violated rules for 'is-<adjective>-'*
	Returns void: Reconsider if included
	Returns boolean: Inappropriate if missing
	Parameter to field: Inappropriate if included
	Replacement phrase:
	'set-<adjective>-*'

// JXTA 2.4.1
public boolean equals(Object obj) {
if (this == obj)
return true;
if (obj instanceof Value)
return equals((Value)obj);
return equals(new Value(obj.toString()));
}
	Semantic Profile:
	Returns boolean, Runtime type-check, Has branches, Multiple returns, Creates regular objects, Creates custom objects, Creates own class objects, Same name call, Method call on parameter
	Violated rules for 'boolean equals(Object...)
	Creates own class objects: Reconsider if included
	Replacement phrase:
	--- None ---

// Java 5 EE SDK
public Iterator iterator()
throws DomainRegistryException {
return new RegistryIterator(this, this);
}
	Semantic Profile:
	Returns reference, No parameters, Return type in name, Returns created object, Creates regular objects, Creates custom objects, Exposes checked exceptions
	Violated rules for 'ref iterator()'
	Exposes checked exceptions: Notify if included
	Replacement phrase:
	'create-<adjective>-<noun>'

// DB Derby 10.2.2.0
public OutputStream setBinaryStream(long val)
throws SQLException {
checkValidity();
synchronized (this.agent_.connection_) {
// Logging code removed.
BlobOutputStream result = new
BlobOutputStream(this, val);
// Logging code removed.
return result;
}
}
	Semantic Profile:
	Returns reference, Reads field, Returns created object, Has branches, Creates regular objects, Exposes checked exceptions, Method call on field
	Violated rules for 'set-<adjective>-'*
	Returns created object: Inappropriate if included
	Replacement phrase:
	'open-*'

Fig. 6: Four notable naming bugs from the corpus.

object that is equal to v. For instance, we can get a true return value if we pass in a suitable String object s. However, if we pass v to the equals method of s, we will get false. Interestingly, we find no appropriate replacement phrase for this method. This is good news, since it makes little sense to rename a broken equals method.

The third example, an iterator method from the class Registry in Java 5 Enterprise Edition, illustrates the problem of overloading and redefining a well-established method name. The heavily implemented Iterable<T> interface defines a method signature Iterator<T> iterator(). Since the signature does not include any checked exceptions, the expectation naturally becomes that iterator methods in general do not expose any checked exceptions — indeed, the compiler will stop implementors of Iterable<T> if they try. However, Registry does not implement Iterable<T>, it simply uses a similar signature. But it is a bad idea to do so, since violating expectations is bound to cause confusion. It is particularly troublesome that the implementation exposes a checked exception, since this is something *iterator* methods practically never do. Note that the replacement phrase makes perfect sense since the method acts as a *factory* that creates new objects.

The final example is a bizarrely named method from DB Derby 10.2.2.0: clearly this is no *setter*! The semantic profile of the method is complicated a bit by the synchronisation and logging code, but for all intents and purposes, this is a factory method of sorts. The essential behaviour is that an object is created and returned to the caller. Creating and returning objects is inappropriate behaviour for methods that match the phrase **set-⟨adjective⟩-***; hence we get a rule violation. The suggested replacement phrase, **open-***, is not completely unreasonable, and certainly better than the original.

5.3 Naming Bug Statistics

We now consider the more general prevalence of naming bugs. Table 7 presents naming bug statistics for all the applications in the corpus, as well as a small number of additional applications. The additional applications are placed beneath a horizontal line near the bottom of the table. For each application, we list the number of methods, the percentage of those methods covered by implementation rules, and the percentage of covered methods violating an implementation rule. We see that the naming bug rates are fairly similar for applications in and outside the corpus, suggesting that the rules can meaningfully be applied to any Java application. It is worth noting that the largest applications (for instance, Java, Eclipse and NetBeans) to some extent have the power to dictate what is common usage. At the same time, such applications are developed by many different programmers over a long period of time, making diversity more likely.

It is important to remember that the numbers really indicate how canonical the method implementations are with respect to the names used. Herein lies an element of conformity as well. The downside is that some applications might be punished for being too "opinionated" about naming. For instance, JUnit 4.2 is written by programmers who are known to care about naming, yet the reported

naming bug rate, 3.50%, is fairly high. We believe this is due to the tension between maintaining the status quo and trying to improve it.

Table 7: Naming bug statistics.

Application	*Methods*	*Covered*	*Buggy*	*Application*	*Methods*	*Covered*	*Buggy*
ANTLR 2.7.6	1641	61.66%	1.18%	ASM 2.2.3	724	45.30%	0.30%
AXIS 1.4	4290	91.35%	1.65%	Ant 1.7.0	7562	89.35%	0.85%
ArgoUML 0.24	13312	81.17%	0.85%	AspectJ 1.5.3	24976	74.41%	1.24%
Avalon 4.1.5	280	82.14%	2.17%	Azureus 2.5.0	14276	78.32%	1.30%
Batik 1.6	9304	85.90%	0.76%	BSF 2.4.0	274	77.37%	0.00%
BeanShell 2.0 Beta	907	74.97%	0.73%	BlueJ 2.1.3	3369	82.13%	1.48%
BluePrints UI 1.4	662	89.57%	0.67%	C3P0 0.9.1	2374	83.06%	1.52%
CGLib 2.1.03	675	80.29%	1.66%	Cactus 1.7.2	3004	87.61%	1.36%
Castor 1.1	5094	91.44%	0.88%	Checkstyle 4.3	1350	76.07%	0.09%
Cobertura 1.8	328	82.92%	1.47%	Commons Codec 1.3	153	79.08%	0.00%
Commons Collections 3.2	2914	77.93%	1.14%	Commons DBCP 1.2.1	823	88.69%	1.09%
Commons Digester 1.8	371	79.24%	0.34%	Commons Discovery 0.4	195	92.30%	0.00%
Commons EL 1.0	277	59.20%	4.87%	Commons FileUpload 1.2	123	91.86%	0.88%
Commons HttpClient 3.0.1	1071	88.98%	1.46%	Commons IO 1.3.1	357	81.23%	5.17%
Commons Lang 2.3	1627	82.72%	1.93%	Commons Modeler 2.0	376	93.35%	1.42%
Commons Net 1.4.1	726	69.69%	1.58%	Commons Pool 1.3	218	71.55%	0.00%
Commons Validator 1.3.1	443	88.03%	1.02%	CruiseControl 2.6	5479	87.18%	0.85%
DB Derby 10.2.2.0	15470	80.08%	2.09%	Dom4J 1.6.1	1645	92.15%	0.39%
Eclipse 3.2.1	110904	81.65%	1.03%	Emma 2.0.5312	1105	82.62%	0.65%
FitNesse	2819	74.49%	2.14%	Ganymed ssh build 209	424	76.65%	1.23%
Genericra	454	86.78%	0.50%	Geronimo 1.1.1	26753	85.28%	0.71%
Google WT 1.3.3	4129	73.40%	1.78%	Groovy 1.0	10237	76.14%	1.01%
HOWL 1.0.2	173	81.50%	1.41%	HSQLDB	3267	86.16%	2.98%
Hibernate 3.2.1	11354	80.47%	2.00%	J5EE SDK	148701	83.56%	1.17%
JBoss 4.0.5	34965	84.69%	0.95%	JDOM 1.0	144	80.55%	0.86%
JEdit 4.3	3330	80.36%	1.30%	JGroups 2.2.8	4165	77.52%	2.04%
JOnAS 4.8.4	30405	81.88%	1.16%	JRuby 0.9.2	7748	76.69%	1.27%
JUnit 4.2	365	62.46%	3.50%	JXTA 2.4.1	5210	86.96%	1.30%
JacORB 2.3.0	8007	71.01%	1.16%	James 2.3.0	2382	79.21%	1.85%
Jar Jar Links 0.7	442	53.84%	0.42%	Java 6 SDK	80292	81.03%	1.16%
JavaCC 4.0	370	77.02%	2.80%	Javassist 3.4	1842	84.03%	1.42%
Jetty 6.1.1	15177	73.54%	1.06%	Jini 2.1	8835	80.00%	1.38%
Jython 2.2b1	3612	72.09%	1.65%	Kawa 1.9.1	6309	65.36%	2.01%
Livewire 4.12.11	12212	81.96%	1.15%	Log4J 1.2.14	1138	83.39%	0.63%
MJC 1.3.2	4957	73.77%	1.72%	MOF	28	100.00%	0.00%
MX4J 3.0.2	1671	85.33%	1.26%	Maven 2.0.4	3686	84.69%	0.86%
Mule 1.3.3	4725	86.79%	1.09%	NetBeans 5.5	113355	87.60%	0.85%
OGNL 2.6.9	502	88.24%	0.45%	OpenJMS 0.7.7 Alpha	3624	85.89%	0.70%
OpenSAML 1.0.1	306	92.48%	1.76%	Piccolo 1.04	559	77.10%	0.46%
PicoContainer 1.3	435	67.81%	1.35%	Polyglot 2.1.0	3521	67.33%	1.64%
Poseidon CE 5.0.1	25739	77.73%	1.19%	Rhino 1.6r5	2238	77.56%	1.67%
Saxon 8.8	6596	73.12%	1.22%	Shale Remoting 1.0.3	96	72.91%	0.00%
Spring 2.0.2	8349	88.05%	1.52%	Struts 2.0.1	6106	88.97%	1.06%
Sun Wireless Toolkit 2.5	20538	80.37%	1.59%	Tapestry 4.0.2	3481	78.71%	0.87%
Tomcat 6.0.7 Beta	5726	88.31%	0.90%	TranQL 1.3	1639	77.85%	1.17%
Trove 1.1b4	3164	82.01%	0.23%	Velocity 1.4	3635	81.62%	0.67%
WSDL4J 1.6.2	651	94.16%	0.00%	XBean 2.0.0	7000	81.10%	1.33%
XML Security 1.3.0	819	86.56%	1.55%	XOM 1.1	1399	77.05%	1.85%
XPP 1.1.3.4	426	84.50%	1.38%	XStream 1.2.1	916	77.83%	0.84%
Xalan-J 2.7.0	14643	81.38%	1.21%	Xerces-J 2.9.0	590	89.15%	0.19%
FindBugs 1.3.6	7688	72.78%	1.42%	iText 2.1.4	4643	85.18%	1.54%
Lucene 2.4.0	2965	74.16%	1.50%	Mockito 1.6	1408	68.32%	1.35%
ProGuard 4.3	4148	45.34%	2.65%	Stripes 1.5	1600	89.31%	2.09%

Where to draw the line between appropriate and inappropriate usage of names is a pragmatic choice, and a trade-off between false positives and false negatives. A narrow range for appropriate usage increases the number of false positives, a broad range increases the number of false negatives. We are not too concerned with false negatives, since our focus is on demonstrating the existence of naming bugs, rather than finding them all. False positives, on the other hand, could pose a threat to the usefulness of our results.

False positives, i.e., that the analysis reports a naming bug that we intuitively disagree with, might occur for the following reasons:

– The corpus may contain noise that leads to rules that are not in harmony with the intuitions of Java programmers.
– Some legitimate sub-use of a commonly used phrase may be deemed inappropriate because the sub-use is drowned by the majority. (Arguably a new phrase should be invented to cover the sub-use.)
– The percentiles used to classify attribute fraction rank (Sect. 3.4) can be skewed.

Whether or not something classifies as a naming bug is subjective. What is *not* subjective, is the fact that all reported issues will be rare, and therefore worthy of reconsideration. To discern false positives from genuine naming bugs, we must rely on our on best judgement. To get an idea of the severity of the problem, we manually investigated 50 reported naming bugs chosen at random. We found that 30% of the reported naming bugs in the sample were false positives, suggesting that the approach holds promise (even though, due to the limited size of the sample, the true false positive rate might be significantly higher or lower). The false positives were primarily *getters* that were slightly complex, but not inappropriately so in our eyes, and methods containing logging code.

5.4 Threats to Validity

There are three major threats to the validity of our results:

– Does the pragmatic view of how meaning is constructed apply to Java programming?
– Is the corpus representative of real-world Java programming?
– Is the attribute model a suitable approximation of the actual semantics of a method?

Our basic assumption is that canonical usage of a method name is also meaningful and appropriate usage; this relates to the pragmatic view that meaning stems from actual use. We establish the meaning of phrases using a crude democratic process of voting. This approach is not without problems. First, it is possible for individual idiosyncratic applications to skew the election. In particular, code generation can lead to problems, since it enables the proliferation of near-identical clones. While we can spot gross examples of this (see Sect. 4), code generation on a smaller scale is hard to detect, and can affect the results for individual phrases. This in turn can corrupt our notion of canonical usage, leading to corrupt rules and incorrect reports of naming bugs. Second, there might be individual applications that use a language that is both richer, more consistent and precise than the one used by the majority. However, the relative uniformity in the distribution of naming bugs seems to indicate that neither of these problems are too severe. Despite these problems, therefore, we believe that the pragmatic view of meaning applies well to Java programming. It is certainly more reasonable to use the aggregated ideas of many as an approximation of meaning than to make an arbitrary choice of a single application's idea.

When aggregating ideas, however, we must assume that the ideas we aggregate are representative. The business journalist Surowiecki argues that diversity of opinion, independence, decentralisation and an aggregation mechanism are the prime prerequisites to make good group decisions [25]. The corpus we use was carefully constructed to contain a wide variety of applications and libraries of various sizes and from many domains. We therefore believe it to fulfil Surowiecki's prerequisites and be reasonably representative of real-world Java programming.

Finally, we consider the suitability of the model for method semantics, which is a coarse approximation based on our knowledge of Java programming. Using attributes to characterise methods has several benefits, in particular that it reduces the practically endless number of possible implementations to a finite set of semantic profiles. Furthermore, the validation of a useful model must come in the form of useful results. As we have seen, the model has helped us identify real naming bugs with what appears to be a relatively low rate of false positives. We therefore believe that the model is adequate for the task at hand.

6 Related Work

Micro patterns, introduced by Gil and Maman [10], are a central source of inspiration for our work. Micro patterns are machine-traceable patterns on the level of Java classes. A pattern is machine-traceable if it can be expressed as a simple formal condition on some aspect of a software module. The presented micro patterns are hand-crafted by the authors to capture their knowledge of Java programming.

In our work, we use hand-crafted machine-traceable attributes to model the semantics of methods rather than classes. The attributes are similar to *fingerprints*, a notion used by the Sourcerer code search engine [1]. According to the Sourcerer website[4], the engine supports three kinds of fingerprint-based search, utilising control flow, Java type and micro pattern information respectively. Ma et al. [16] provide a different take on the task of searching for a suitable software artefact. They share our assumption that programmers usually choose appropriate names for their implementations, and therefore use identifier information to index the Java API for efficient queries.

Overall, there seems to be a growing interest in harnessing the knowledge embedded in identifiers. Pollock et al. [20] introduce the term *Natural Language Program Analysis* (NLPA) to signify program analysis that exploits natural language clues. The analysis has been used to develop tools for program navigation and aspect mining [23, 22]. The tools exploit the relationship between natural language expressions in source code (identifiers and comments) and information about the structure of the code.

Singer and Kirkham [24] investigate which type names are used for instances of micro patterns in a large corpus of Java applications. More precisely, the *suffixes* of the actual type names are used (the last *fragment* of the name in our

[4] `http://sourcerer.ics.uci.edu/`

terminology). The empirical results indicate that type name suffixes are indeed correlated to the presence of micro patterns in the code.

Caprile and Tonella [4] analyse the structure of function identifiers in C programs. The identifiers are decomposed into fragments that are then classified into seven lexical categories. The structure of the function identifiers are further described by a hand-crafted grammar.

Lawrie et al. [13] study the quality of identifiers in a large corpus of applications written in several languages. An identifier is assumed to be of high quality if it can be composed of words from a dictionary and well-known abbreviations. This is a better quality indicator than mere uniformity of lexical syntax, but does not address the issue of *appropriateness*. Deißenböck and Pizka [6] develop a formal model for identifier quality, based on *consistency* and *conciseness*. Unfortunately, this model requires an expert to perform manual mapping between identifiers and domain concepts.

Reiss [21] proposes an automatic approach for finding unusual code. The assumption is that unusual code is potentially problematic code. The approach works by mining common syntactic code patterns from a corpus of applications. Unusual code is code that is not covered by such patterns. Hence we see that there are similarities to our work, both in the assumption and the approach. A main difference is that we define unusual code in the context of a given method phrase.

7 Conclusion

Natural language expressions get their meaning from how and when they are used in practice. Deviation from normal use of words and phrases leads to misunderstanding and confusion. In the context of software this is particularly bad, since precise understanding of the code is paramount for successful development and maintenance. We have therefore coined the term *naming bug* to describe unusual aspects of implementations for a given method name. We have presented a practical approach to *debugging method names*, by offering assistance both in finding and fixing naming bugs. To find naming bugs, we use name-specific implementation rules mined from a large corpus of Java applications. Naming bugs can be fixed either by changing the implementation or by using a different method name; for the latter task, we have also shown an approach to provide automatic assistance. To demonstrate that method name debugging is useful, we have applied the rules to uncover naming bugs both in the corpus itself and in other applications.

In this and previous work, we have exploited the fact that there is a shared vocabulary of terms and phrases, *Java Programmer English* [12], that programmers use in method names. In the future, we would like to investigate the adequacy of that vocabulary. In particular, there might be terms or phrases that are superfluous, while others are missing, at least from the common vocabulary of Java programmers. We know that there exists verbs (for instance *create* and *new*) that seem to be used almost interchangeably in method names. Our results

reveal hints of this, by finding a shorter semantic distance between phrases that use such verbs. By analysing the corresponding method implementations, we could find out whether there are subtle differences in meaning that warrant the existence of both verbs in Java Programmer English. If not, it would be beneficial for Java programmers to choose one and eliminate or redefine the other. There are also verbs (and phrases) that are imprecise, in that they are used to represent many different kinds of implementations. For instance, the ubiquitous *getter* is much less homogenous than one might expect [11], indicating that it has a wide variety of implementations. It would be interesting to see if the verbs are simply used as easy resorts when labelling more or less random chunks of code, or if there are legitimate, identifiable sub-uses that would warrant the invention of new verbs. Or it might be that a minority of the Java community already has invented the proper verbs, and that they should be more widely adopted to establish a richer, more expressive language for all Java programmers to use.

Acknowledgements. We thank Jørn Inge Vestgården, Wolfgang Leister and Truls Fretland for useful comments and discussions, and the anonymous reviewers for their thoughtful remarks.

References

[1] S. K. Bajracharya, T. C. Ngo, E. Linstead, Y. Dou, P. Rigor, P. Baldi, and C. V. Lopes. Sourcerer: a search engine for open source code supporting structure-based search. In P. L. Tarr and W. R. Cook, editors, *OOPSLA Companion*, pages 681–682. ACM, 2006.

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

[3] J. Bloch. *Effective Java*. Prentice Hall, 2008.

[4] B. Caprile and P. Tonella. Nomen est omen: Analyzing the language of function identifiers. In *Proceedings of the Sixth Working Conference on Reverse Engineering (WCRE 1999), 6-8 October 1999, Atlanta, Georgia, USA*, pages 112–122. IEEE Computer Society, 1999.

[5] E. Collar and R. Valerdi. Role of software readability on software development cost. In *Proceedings of the 21st Forum on COCOMO and Software Cost Modeling, October 2006, Herndon, VA.*, 2006.

[6] F. Deißenböck and M. Pizka. Concise and consistent naming. In *Proceedings of the 13th IEEE International Workshop on Program Comprehension (IWPC 2005)*, pages 97–106. IEEE Computer Society, 2005.

[7] M. A. Eierman and M. T. Dishaw. The process of software maintenance: a comparison of object-oriented and third-generation development languages. *Journal of Software Maintenance and Evolution: Research and Practice*, 19(1):33–47, 2007.

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

[9] G. Frege. On sense and reference. In P. Geach and M. Black, editors, *Translations from the Philosophical Writings of Gottlob Frege*, pages 56–78. Blackwell, 1952.

[10] J. Gil and I. Maman. Micro patterns in Java code. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2005), October 16-20, 2005, San Diego, CA, USA*, pages 97–116. ACM, 2005.

[11] E. W. Høst and B. M. Østvold. The programmer's lexicon, volume I: The verbs. In *Proceedings of the Seventh IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2007)*, pages 193–202, Washington, DC, USA, 2007. IEEE Computer Society.

[12] E. W. Høst and B. M. Østvold. The Java programmer's phrase book. In *Proceedings of the 1st International Conference on Software Language Engineering (SLE 2008)*. Springer, 2008.

[13] D. Lawrie, H. Feild, and D. Binkley. Quantifying identifier quality: An analysis of trends. *Journal of Empirical Software Engineering*, 12(4):359–388, August 2007.

[14] 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.

[15] B. Liblit, A. Begel, and E. Sweeser. Cognitive perspectives on the role of naming in computer programs. In *Proceedings of the 18th Annual Psychology of Programming Workshop*, Sussex, United Kingdom, September 2006. Psychology of Programming Interest Group.

[16] H. Ma, R. Amor, and E. D. Tempero. Indexing the Java API using source code. In *Australian Software Engineering Conference*, pages 451–460. IEEE Computer Society, 2008.

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

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

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

[20] L. L. Pollock, K. Vijay-Shanker, D. Shepherd, E. Hill, Z. P. Fry, and K. Maloor. Introducing natural language program analysis. In *Proceedings of the 7th ACM Workshop on Program Analysis for Software Tools and Engineering (PASTE 2007), San Diego, California, USA, June 13-14, 2007*, pages 15–16. ACM, 2007.

[21] S. P. Reiss. Finding unusual code. In *Proceedings of the 23rd IEEE International Conference on Software Maintenance (ICSM 2007)*, pages 34–43. IEEE Computer Society, 2007.

[22] D. Shepherd, Z. P. Fry, E. Hill, L. Pollock, and K. Vijay-Shanker. Using natural language program analysis to locate and understand action-oriented concerns. In *Proceedings of the 6th international conference on Aspect-oriented software development (AOSD 2007)*, pages 212–224, New York, NY, USA, 2007. ACM.

[23] D. Shepherd, L. L. Pollock, and K. Vijay-Shanker. Towards supporting on-demand virtual remodularization using program graphs. In *Proceedings of the 5th International Conference on Aspect-Oriented Software Development (AOSD 2006)*, pages 3–14. ACM, 2006.

[24] J. Singer and C. Kirkham. Exploiting the correspondence between micro patterns and class names. In *Proceedings of the Eight IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2008)*, pages 67–76. IEEE Computer Society, 2008.

[25] J. Surowiecki. *The Wisdom of Crowds*. Anchor, 2005.

[26] I. H. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann, 2nd edition, 2005.

[27] L. Wittgenstein. *Philosophical Investigations*. Prentice Hall, 1973.