Seventh IEEE International Working Conference on Source Code Analysis and Manipulation

The Programmer's Lexicon, Volume I: The Verbs

Einar W. Høst, Bjarte M. Østvold
Norwegian Computing Center

Email: {einarwh,bjarte}@nr.no

Abstract

Method names make or break abstractions: good ones communicate the intention of the method, whereas bad ones cause confusion and frustration. The task of naming is subject to the whims and idiosyncracies of the individual since programmers have little to guide them except their personal experience. By analysing method implementations taken from a corpus of Java applications, we establish the meaning of verbs in method names based on actual use. The result is an automatically generated, domain-neutral lexicon of verbs, similar to a natural language dictionary, that represents the common usages of many programmers.

1. Introduction

At the heart of programming is *abstraction*, the creation and naming of a set of behaviours — an implementation — to form an aggregated behaviour that can be invoked by referring to the name. The wonderful thing about abstraction is that it scales: we can build new abstractions using those we have previously created. The crucial part of abstraction is to have the name reflect the semantics of the implementation. Failure in this regard is catastrophic, as a single faulty abstraction contaminates all abstractions built on top of it.

We conclude that the problem of *naming* is vital to the task of programming. The programmer must constantly ask of herself:

- What names should I use?
- Is this a good name for the behaviour?
- Will other programmers understand the meaning of this name?
- How can I be sure that I've used a name correctly?

If we were dealing with words in natural language, we might consult a dictionary to help answer these questions. Lacking such a tool, we turn instead to the implementation that the name is supposed to represent. Consider, for instance, the following simple Java method, where the name has been omitted:

```
Person ___() {
  return p;
}
```

In spite of the missing name, we immediately notice that it is a *getter*, that is, a method whose name should start with *get*, and which returns the value of a local field.

Adding a parameter and a loop, we get:

```
Person ___(String id) {
  for (Person p : persons) {
    if (p.getID().equals(id)) {
      return p;
    }
  }
  return null;
}
```

Now we would consider it sloppy naming, if not downright wrong, if the method name started with *get*; here we are clearly trying to *find* something.

While trivial, these examples illustrate something important: there are *clues* in the implementation that can be used to indicate whether or not a name is suitable. This leads us to formulate the problem investigated in this paper: *Can we create a semantics which captures our common interpretation of method names?*

The essence of our approach is to encode the implementation of a method by means of *semantic attributes*; properties that a given implementation may or may not possess. A simple example is whether or not the implementation contains a loop. This encoding can then be used to characterise the name of the method, by aggregating all the encodings that share the same name. The characterisation constitutes a sort of "usage semantics" for the method name. Clearly, this is radically different from the formal semantics of the program itself. In a sense, it is the difference between

0-7695-2880-5/07 $25.00 © 2007 IEEE
DOI 10.1109/SCAM.2007.18

the semantics of the informal *programmer* language on one hand and the formal *programming* language on the other.

For simplicity, we consider only the domain-neutral, action-oriented initial part of a Java method name. Typically, this is a *verb*. Hence, if the full method name is `findPersonByID`, we abstract the name to be simply *find*. In other words, we investigate the properties of *getters*, *setters*, *finders*, *closers*, *adders* and so forth. These names indicate the basic *actions* performed by Java methods. By linking these verbs to the attributes of the method implementations, we are able to create the missing dictionary of "programmer English". We call this dictionary *The Programmer's Lexicon*.

The contributions of this paper are:

- A set of attributes, defined on Java byte code, that can be used to characterise the implementation of a method (Section 4).
- A definition of the *usage semantics* of a name by means of the distribution of attribute-value combinations in a corpus (Section 2.3), establishing a formal relationship between method names and implementations.
- A measure for the precision of a name in programmer English (Section 3.3), based on a notion of entropy related to the semantics.
- A technique for comparing names, based on comparing their semantics (Section 3.4).

We use our techniques to analyse a large software corpus (Section 5), and explain the results by investigating some notable examples (Section 6).

The contributions are manifest in *The Programmer's Lexicon*, an automatically generated description of domain-neutral verbs often used in Java programming. The lexicon can be found in the appendix.

2. Definitions

2.1. Preliminaries

An *attribute* has an attribute name a and a binary value b, that is, the value is 0 or 1. For simplicity, we consider an attribute and its name as the same. An *object* o has three features: two symbols and a set of attributes $a_1, \ldots, a_m$ with values $v_1, \ldots, v_m$. The symbols are a unique *fingerprint* u and a *name* n. We use fingerprints for technical purposes and never consider their actual values. Unique fingerprints ensure that a set made from arbitrary objects $o_1, \ldots, o_k$ always has k elements, that is, they prevent that several set elements collapse into one.

A *corpus* $\mathcal{C}$ is a set of objects with the same attributes. We sometimes leave $\mathcal{C}$ implicit when there is no risk of confusion. There are two fundamental ways of dividing a corpus into parts: group objects with the same name together, or group objects with the same attribute values together. We need both.

2.1.1. Objects with same name. The *n-corpus* of $\mathcal{C}$, denoted $\mathcal{C}(n)$, is the set of all objects from corpus $\mathcal{C}$ that have the same name n. The *cardinality* of name n in $\mathcal{C}$, denoted $|n|_{\mathcal{C}}$, is defined as follows:

$$|n|_{\mathcal{C}} \stackrel{\text{def}}{=} |\mathcal{C}(n)|$$

where $|\mathcal{C}(n)|$ is the cardinality of set $\mathcal{C}(n)$. The *relative frequency* of an attribute a with respect to a name n, denoted $\xi_a(n)$, is the fraction of objects in $\mathcal{C}(n)$ that has attribute a set to value 1.[1]

2.1.2. Objects with same attribute values. If two objects o, o' have the same values for all attributes we say that they are *attribute-value identical*, denoted $o \simeq o'$. Note that this relation ignores the fingerprint and name of an object. Using relation $\simeq$ we can divide a corpus $\mathcal{C}$ into a set of equivalence classes $\mathrm{EC}(\mathcal{C}) = [o_1]_{\mathcal{C}}, \ldots, [o_k]_{\mathcal{C}}$, where $[o]_{\mathcal{C}}$ is defined as:

$$[o]_{\mathcal{C}} \stackrel{\text{def}}{=} \{o' \in \mathcal{C} \mid o' \simeq o\}.$$

We simplify the notation to $[o]$ when there can be no confusion about the interpretation of $\mathcal{C}$.

By definition the equivalence classes of a corpus are disjoint — each object belongs to exactly one equivalence class. The cardinality $|[o]_{\mathcal{C}}|$ is the number of distinct objects in $\mathcal{C}$ that are equivalent to o by relation $\simeq$. The sum of the cardinalities of the equivalence classes equals the cardinality of $\mathcal{C}$,

$$|[o_1]| + \cdots + |[o_k]| = |\mathcal{C}|.$$

2.2. Distribution and entropy

We repeat some information-theoretical concepts related to entropy [2]. Let X be a discrete random variable with alphabet χ and probability mass function $p(x) = Pr\{X = x\}, x \in \chi$. Since a) for all $i = 1, \ldots, k$ it holds that $0 \leq p(x_i) \leq 1$; and b) $\sum_{i=1}^{k} p(x_i) = 1$, we have that $p(x_1), \ldots, p(x_k)$ form a *probability distribution*.

The *Shannon entropy* H of X can then be defined as:

$$H(X) \stackrel{\text{def}}{=} -\sum_{x \in \chi} p(x) \log_2 p(x)$$

1. Gil and Maman call $\xi_a(n)$ the *prevalence* of 'pattern' a [6].

where we assume $0 \log_2 0 = 0$.

The entropy of a distribution measures the uncertainty of a random variable having that distribution. Alternatively, it measures the expected number of bits required to represent an event from the distribution.

Next we define the entropy of a corpus, and based on this, the entropy of a name. Let the probability mass function $p([o])$ of corpus $\mathcal{C}$ be defined as

$$p([o]) \stackrel{\text{def}}{=} \frac{|[o]|}{|\mathcal{C}|}, \quad [o] \in \text{EC}(\mathcal{C}).$$

By the definitions of $[o]$ and cardinality, it follows that a corpus has a probability distribution $p([o_1]), \ldots, p([o_k])$. Thus we can define the entropy of corpus $\mathcal{C}$ as

$$H(\mathcal{C}) \stackrel{\text{def}}{=} H\big(p([o_1]), \ldots, p([o_k])\big).$$

Since $\mathcal{C}(n)$ also has a probability distribution, we have that a name has an entropy, denoted $H(n)$, defined as the entropy of $\mathcal{C}(n)$,

$$H(n) \stackrel{\text{def}}{=} H\big(\mathcal{C}(n)\big).$$

2.3. The Usage Semantics of Names

We define the *usage semantics* of a name n, written $[\![n]\!]$, in terms of $\mathcal{C}(n)$ as follows:

$$[\![n]\!] \stackrel{\text{def}}{=} \big\{([o], |[o]|) \mid [o] \in \text{EC}(\mathcal{C}(n))\big\},$$

where $([o], |[o]|)$ is the pair consisting of the equivalence class $[o]$ and the cardinality of that class.

Thus $[\![n]\!]$ reflects all the ways in which n is used in $\mathcal{C}(n)$, as well as the number of times it is used in each way. We can visualise $[\![n]\!]$ by drawing a vertical bar for each equivalence class $[o]$ in the probability distribution of $\mathcal{C}(n)$. We refer to this visualisation as the *distribution diagram* for n (see Section 6).

Finally, we define a function S that yields a set of equivalence classes which each cover at least a fraction q of the objects in $\mathcal{C}(n)$:

$$S(n, q) \stackrel{\text{def}}{=} \big\{([o], |[o]|) \mid ([o], |[o]|) \in [\![n]\!] \wedge p([o]) \geq q\big\}$$

We call this a *spike set*. It has a straightforward interpretation in light of distribution diagrams, in that the most prominent equivalence classes reveal themselves as spikes in the diagrams.

3. Approach to Name Analysis

The names we consider in this paper are abstractions of the real method names used in Java programs. The aim is to capture the essence of the common names — typically verbs — used to denote the *actions* performed by Java methods. For instance, the concrete method names `open`, `openConnection` and `openFile` will all be considered instances of the abstract name *open*. Hence a *name* is an abstraction that will typically represent many concrete methods.

We investigate the name abstraction by looking at what is being abstracted; that is, we distill information from analysing the implementation of each method. In doing so, we apply a corpus-based *usage semantics* for names, in that the meaning is determined by the actual use of the name in a large software corpus. This is similar to how the semantics of words in natural language is established.

Of particular interest to us is the *precision* of the name, that is, how clearly the abstraction indicates the semantic content of the method, a *description* of the name, that is, what the typical semantic content is, and a *comparison* of the name to other names, in particular to find names that are similar or related in some way.

A problem not addressed in our current work is that of *polysemy*: the same name may have more than one meaning. If present, polysemy will manifest itself indirectly as lowered precision in the characterisation of the name, as well as a potentially skewed description of the name itself.

3.1. Restricting the Set of Names

Since the set of names used in programming is potentially unbounded, we device an algorithm for establishing the set of *common names* based on all the names in the corpus.

To ameliorate the effect of any idiosyncracies in large software projects (for instance, Sun's Java API), we sort the corpus of applications alphabetically, mechanically divide it into k subcorpora, and choose the n most frequent names in each subcorpus. Constructing the intersection of the k sets yields a set of N names, where $|N| \leq n$. This is the set of *common names*. The set of objects with common names is denoted $\mathcal{C}_{com}$. We use this set as the data material from which we establish semantic similarities and dissimilarities.

For the sake of brevity, we focus our investigation on a subset of the $m < |N|$ most frequent among the common names. This is the set of names presented in *The Programmer's Lexicon*. We write $\mathcal{C}_{lex}$ for the corresponding corpus of objects.

The concrete values used in our analysis are $k = 5$ subcorpora with $n = 150$ candidate names from each subcorpus, yielding $|N| = 100$ common names. The number of names in the lexicon is restricted to $m = 40$.

Percentile	*Group name*
$< 5\%$	Low extreme
$< 25\%$	Low
25% - 75%	Unlabelled
$> 75\%$	High
$> 95\%$	High extreme

Table 1: Quantile groups for attribute values.

3.2. Describing Names

As is the case for natural language, it makes little sense to describe a name in isolation; a symbol requires the contrast of other symbols to become meaningful. We therefore wish to say that the relative frequency of an attribute on a name, $\xi_a(n)$, is high or low compared to that of all other names.

For a given attribute a, the relative frequencies $\xi_a(n)_i$ for all names $n_i \in N$ are distributed within the boundaries $0 \leq \xi_a(n)_i \leq 1$. We divide this distribution into five named groups, based on the 5%, 25%, 75% and 95% percentiles of relative frequencies, as shown in Table 1. Each name then becomes associated with a certain group for a, depending on the value for $\xi_a(n)$: the 5% of names with the lowest relative frequencies end up in the "low extreme" group, and so forth.

Taken together, the group memberships for attributes $a_i, \ldots, a_k$ becomes an abstract characterisation of a name, which can be used to generate a description of it.

3.3. Measuring the Precision of Names

Intuitively, precision denotes how consistently a name refers to the same *thing* or combination of things. In our context, this translates to *attribute value combinations*. If a name n tends to indicate the same combinations of values for the objects in $\mathcal{C}(n)$, we think of it as precise. In other words, the more dependent the attributes are on each other for a name n, the more precise n is.

Since entropy is a measure of how independent the attributes are, we can use entropy to measure the precision of each name. A precise name has a low degree of entropy, an imprecise name a high degree. However, *low* and *high* are relative notions; hence, a name can only be precise or imprecise compared to other names. We therefore base our characterisation on quartiles: the names with entropy in the lowest quartile are deemed *precise*, in the highest quartile *imprecise*.

3.4. Comparing and Relating Names

A basic assumption for our work is that no name is completely arbitrary or imprecise. For any name, then, some equivalence classes will consist of more objects than others. These equivalence classes can be thought of as the distinguishing traits of the name. We exploit this fact to compare individual names, with the aim of characterising the relationship between them. This allows us to conveniently ignore inevitable variations in precision and nuance between names, and focus on the essential similarities or differences.

We use the *spike sets* $S(n_1, q)$ and $S(n_2, q)$ (see Section 2.3) to characterise two names n_1 and n_2 as being:

- *Similar*, in which case $S(n_1, q) = S(n_2, q)$.
- *Generalisations* or *specialisations* of each other. We say that n_1 generalises n_2 (and, conversely, that n_2 specialises n_1) if $S(n_1, q) \subset S(n_2, q)$.
- *Somewhat related*, when $S(n_1, q) \cap S(n_2, q) \neq \emptyset$.

The value for q must be set based on human judgement — we simply choose the value that seems to yield the best results: $q = 0.1$.

4. The Attribute Catalogue

Gil and Maman [6] define the term *traceable pattern* as "a simple formal condition on the attributes, types, name and body of a software module and its components." Here formal means that a program can check if a module matches a pattern or not. The term module includes packages, classes, methods, procedures, and fragments of code, code attributes or names. Design patterns [5] are not traceable: they cannot be recognised mechanically. A traceable pattern on a method or procedure is called a *nano pattern*.

We do not propose nano patterns here; rather, we define a set of *traceable attributes* that could be used as building blocks for creating such patterns. An attribute is *traceable* if its value can be determined mechanically. We also require that the attributes be *independent*, in the sense that the value of an attribute cannot be derived logically from another.

We define our attributes in terms of formal conditions on the byte code. We have chosen to analyse byte code because it is easily available both for open source and commercial applications, and because we are then guaranteed to analyse the actual code that runs.

The attributes are listed in Table 2. For explanations of the terms used in the formal definitions, see Lindholm and Yellin [9]. The selection is based on our experience as Java programmers. The attributes are meant to indicate the basic, generic behaviours of a method implementation. For instance, *field writer* indicates that the method alters the state of an object, *same name call* hints at recursion or delegation, and so forth.

Name	*Formal definition*
Returns void	The *return descriptor* is ***V***.
No parameters	The list of *parameter descriptors* is empty.
Field reader	*GETFIELD* or *GETSTATIC* instruction.
Field writer	*PUTFIELD* or *PUTSTATIC* instruction.
Contains loop	Jump instructions that allow for instructions to be executed more than once in the same method invocation.
Creates object	*NEW* instruction.
Throws exception	*ATHROW* instruction.
Type manipulator	*INSTANCEOF* or *CHECKCAST* instruction.
Local assignment	One of the *STORE* instructions (for instance, *ISTORE*).
Same name call	Calls a method of the same name.

Table 2: The attribute catalogue.

4.1. Critique of the Catalogue

Our current choice of attributes is somewhat arbitrary, in the sense that it rests on our intuitions about what distinguishes methods from each other. A more structured approach would be to use the marginal entropy [6] of individual attributes to select from a pool of candidate attributes those that provide the best separation power. That way, we would rely less on our own preconceptions.

Furthermore, the quality of attributes is limited by the sophistication of our current analysis. Using simple data flow analysis [10], for instance, we could define more poignant attributes such as "return value stems from field", or "parameter value is written to field".

5. The Corpus of Java Programs

We introduce some informal terms to aid in the discussion of our data set. By *application* we mean a compiled Java application having an intended use. Applications may range widely in domain and complexity, from the lithe JUnit testing framework to the massive JBoss Application Server. A software *collection* is a set of applications. A *corpus* is large collection chosen deliberately to cover a spectrum of intended purposes, to ensure that it is representative of all kinds of applications.

We had two main goals when gathering applications for the software corpus: we wanted it to be as large as possible, and we wanted it to consist of applications that are commonplace or well-known.

We identified several groups of applications to help balance the corpus, and to make sure it covered a wide range of domains: desktop applications, programmer tools, languages, language tools, middleware, servers, software development kits, XML tools and common utilities. Note that this grouping was not intended to be an exhaustive taxonomy for applications, but rather to act as a skeleton to span the extent of our corpus. The resulting list of applications is presented in Table 3.

Applications	
Desktop applications	
ArgoUML 0.24	JEdit 4.3
Azureus 2.5.0	LimeWire 4.12.11
BlueJ 2.1.3	NetBeans 5.5
Eclipse 2.3.1	Poseidon CE 5.0.1
Programmer tools	
Ant 1.7.0	FitNesse
Cactus 1.7.2	JUnit 4.2
Cobertura 1.8	Maven 2.0.4
CruiseControl 2.6	Velocity 1.4
Languages	
BeanShell 2.0b	Jython 2.2b1
Groovy 1.0	Kawa 1.9.1
JRuby 0.9.2	Rhino 1.6r5
Language tools	
ANTLR 2.7.6	MJC 1.3.2
ASM 2.2.3	JavaCC 4.0
AspectJ 1.5.3	Polyglot 2.1.0
BCEL 5.2	
Middleware and frameworks	
AXIS 1.4	PicoContainer 1.3
Jini 2.1	Spring 2.0.2
JXTA 2.4.1	Struts 2.0.1
OpenJMS 0.7.7a	Tapestry 4.0.2
Mule 1.3.3	
Servers	
Geronimo 1.1.1	Jetty 6.1.1
James 2.3.0	JOnAS 4.8.4
JBoss 4.0.5	Tomcat 6.0.7b
Software development kits	
Google Web Toolkit 1.3.3	Java 6 SDK
Java 5 EE SDK	Sun Wireless Toolkit 2.5
XML tools	
Castor 1.1	Xerces-J 2.9.0
JDOM 1.0	XOM 1.1
Saxon 8.8	
Common utilities	
Hibernate 3.2.1	Log4J 1.2.14

Table 3: Original list of corpus applications.

Since applications are rarely built from scratch, they often contain dependencies upon other bits and pieces of software, ranging from applications to libraries to individual class files. Hence, the corpus is littered with all kinds of additional applications that we did not originally plan to include.

In principle, we would like to identify, separate and label all the different applications in the corpus. In practice, this task is infeasible due to the multitude of applications and versions, and the myriad ways they can be combined and intertwined. Instead, we chose to eliminate JAR files that contained many classes that collide with classes in other JAR files, that is, when the classes had the same fully qualified name.

The pruned corpus contains:

- 1004 JAR files
- 190572 class files

- 1384205 non-constructor methods
- 157779 omitted methods
- 1226426 included methods

We enforce rather strict qualifications for the methods to be included in the corpus. In addition to ignoring constructors, we also omit all synthetic methods. Furthermore, we demand that method names follow the standard camel-case convention for Java, use letters or digits only, and consist of more than a single character. For instance, the method name `getParser()` is included, whereas `get_parser()`, `getParser$1()` and `f()` are all omitted. The primary rationale for this strictness is that the camel-case convention is so well-established and well-known that we consider it a sign of noise when it is not followed. For instance, it might indicate that the code was generated.

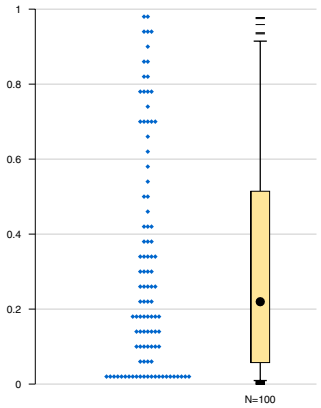


Figure 1: Distribution of relative frequencies for the *no parameters* attribute.

6. Experimental Results

We perform a fully automated analysis of the software corpus. The output of our analysis is summarised in *The Programmer's Lexicon*, printed in the appendix.

As an example illustrating both how the lexicon is constructed and how to read it, we look at the name *get* and its closest neighbours semantically. Note that the observations we make merely mimic those made mechanically by our analysis software. The name *get* is interesting because it is by far the most common one; nearly a third of all Java methods in the corpus are *get*-methods.

The Programmer's Lexicon defines *get* as follows:

> **get.** The most common method name. Methods named *get* often read state and have no parameters, and rarely return void, call methods of the same name, manipulate state, use local variables or contain loops. A similar name is *has*. Specialisations of *get* are *is* and *size*. A somewhat related name is *hash*.

A Java programmer should not be very surprised by this description: *get* methods tend to be short and simple functions that read object state. That methods starting with the name *has*, *is*, *size* and *hash* fit more or less the same description also matches intuition.

The entry for each name is generated by combining several pieces of information; the frequency and entropy of the name, an account of how its usage semantics compare to that of other names, and the spikes showing the most common attribute combinations for the name.

The description of the characteristics of methods with a given name is based on how the relative frequencies of attributes compare to methods with other names in the same corpus. For instance, *get* has a relative frequency of approximately 0.694 for the *no parameters* attribute. The distribution plot in Figure 1 shows how this compares to other names. Each dot represents one of the *common names*. We can see that 0.694 places *get* between the 75% and 95% percentiles, which leads us to characterise its score as *high*, see Table 1. For a mapping between quantile groups and the words used in the lexicon, see Table 5. It turns out that *get* has no attribute frequencies in the extremal groups.

Some significant entropy values are listed in Table 4. We see that *get* has a higher degree of entropy than we might have anticipated. This implies that *get*-methods are not always simple field-retriever functions. Investigating the table a bit further, we notice that there are names such as *load*, with greater entropy than the corpus as a whole. The reason is that the entropy of the corpus is dominated by the most common names; again, nearly a third of all methods are *getters*. Apart from that, the most surprising entry is that of *parse*, which appears to be much more precise than we would have guessed. The explanation is that the Apache XmlBeans project, which is distributed as part of Geronimo, contributes more than 3000 near-identical *parse* methods. Presumably these have been generated. Unfortunately, there is no simple way to automatically discover generated code.

For each name n, we visualise the probability distribution for $\mathcal{C}(n)$ by means of a distribution diagram. The height of each vertical bar is $p([o])$, meaning that the y-axis signifies the fraction of objects belonging to an equivalence class. Since we use ten binary attributes, we have 2^{10} equivalence classes, yielding a resolution of 1024 on the x-axis.

Corpus	*Entropy*	Comment
$\mathcal{C}$	6.8893	All names
$\mathcal{C}_{com}$	6.7591	Common names
$\mathcal{C}_{lex}$	6.5931	Lexicon names
$\mathcal{C}(size)$	2.5343	Most precise name
$\mathcal{C}(load)$	7.4798	Least precise name
$\mathcal{C}(get)$	4.9966	Most common name
$\mathcal{C}(hash)$	3.5616	
$\mathcal{C}(has)$	4.1766	
$\mathcal{C}(is)$	4.2318	
$\mathcal{C}(parse)$	3.6886	Suspiciously low

Table 4: Significant entropy values.

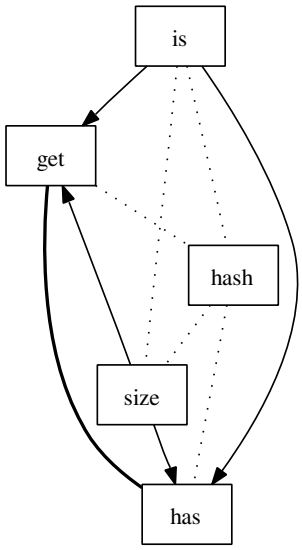


Figure 3: The relationships between *get* and associated names.

Figure 2 shows the distributions for *get* and its semantic neighbours[2]. For clarity, the relationship between the names is also illustrated in Figure 3, where similar names are connected with a bold line, specialisations point to generalisations, and somewhat related names are connected with a dotted line.

Recall that an equivalence class is included in the spike set $S(n, q)$ for a name n if it accounts for at least a fraction $q = 0.1$ of the objects in $\mathcal{C}(n)$. As we can see from Figures 2a and 2b, the spikes of *get* and *has* that are higher than 0.1 correspond perfectly. This leads us to label the names as "similar".

Figure 2c, on the other hand, reveals that *size* has a spike that *get* does not, but not vice versa. The spike indicates a significant, specialised use; hence *size* is a specialisation of *get*. The additional spike for *size*, at position $x = 580$, represents a group of methods that read state, have no parameters, and also call other methods named *size*. This matches our intuition of what a *size* method might look like, and it also makes sense that *get* methods are not like that.

Finally, the lexicon entry says that *get* is somewhat related to *hash*. This stems from the fact that they have a spike in common (where only the attributes *reads state* and *no parameters* are set), but also that they both have spikes that are not shared by the other. To see this requires a little scrutiny of Figures 2a and 2d. The bar at position $x = 512$ for *get* represents 10.8% of all *getters*, whereas the corresponding bar for *hash* represents merely 3.1% of *hash* methods. It is more obvious that *hash* has at least one spike not shared by *get*; namely, a spike at position $x = 580$ similar to the one that differentiates *size* from *get*.

6.1. Exploring Nuances with a Larger Lexicon

The Programmer's Lexicon has been kept tiny for the sake of brevity and readability. Our approach can easily be used to generate a much larger and more detailed lexicon for the same corpus, allowing us to investigate more subtle nuances between names. The only prerequisite is that the cardinality for each name, $|n|_{\mathcal{C}}$, must be large enough for the analysis to be meaningful.

In such a case a printed lexicon might become unwieldy, but relationships can still be investigated meaningfully using graphs. An example graph of *dispose* and related words, taken from a lexicon generated with $n = 200$ candidate names and $m = 100$ chosen names (see Section 3.1), is shown in Figure 4. The corpus is the same that was used to generate *The Programmer's Lexicon*.

7. Related Work

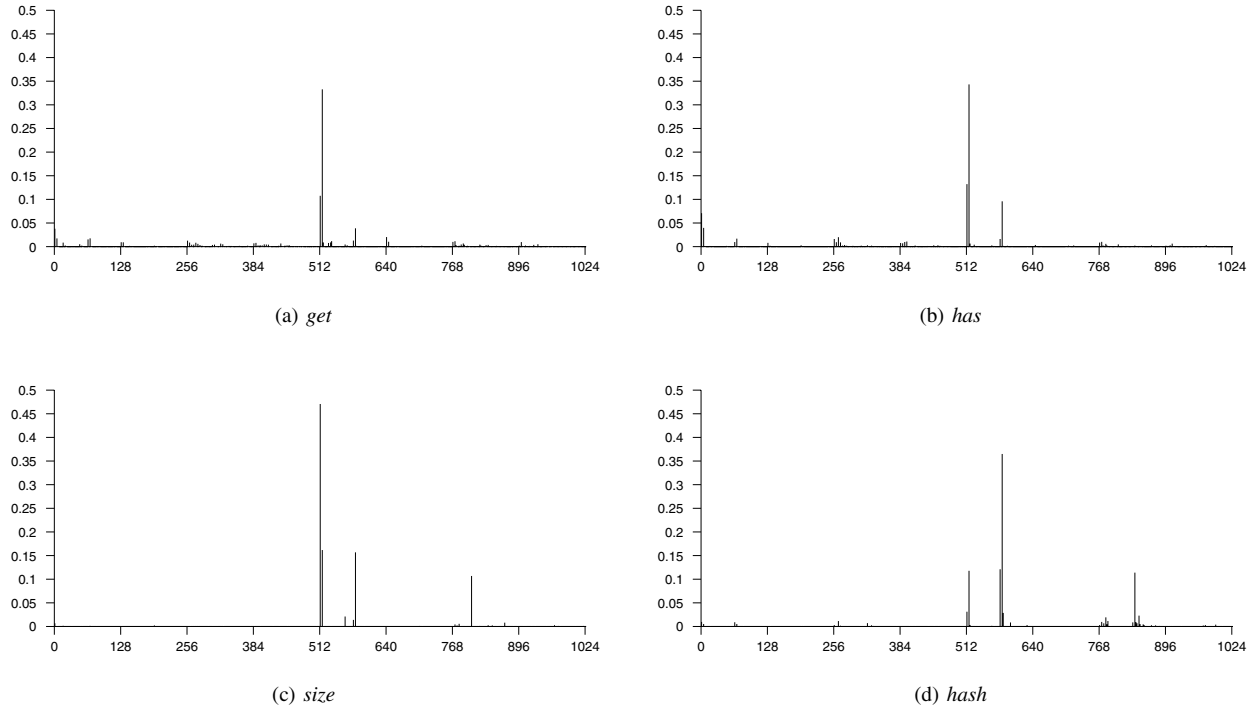
The importance of names is well-understood by industry practitioners. In blogs and articles, fairly sophisticated discussions of names are carried out for instance by Martin Fowler, investigating the confusion caused by homonyms in source code[3], and Steve Yegge, complaining about the emphasis put on nouns over verbs in Java[4].

Among researchers, names have primarily been analysed in the context of readability and program comprehension. Deißenböck and Pizka [3] define precise rules for the conciseness and consistency of names based on a manually constructed formal model. Lawrie

2. Except *is*, which is omitted because it resembles *size*.

3. http://martinfowler.com/bliki/TypeInstanceHomonym.html

4. http://steve-yegge.blogspot.com/2006/03/execution-in-kingdom-of-nouns.html



(a) *get* (b) *has*

(c) *size* (d) *hash*

Figure 2: Distribution of *get* and its semantic neighbours.

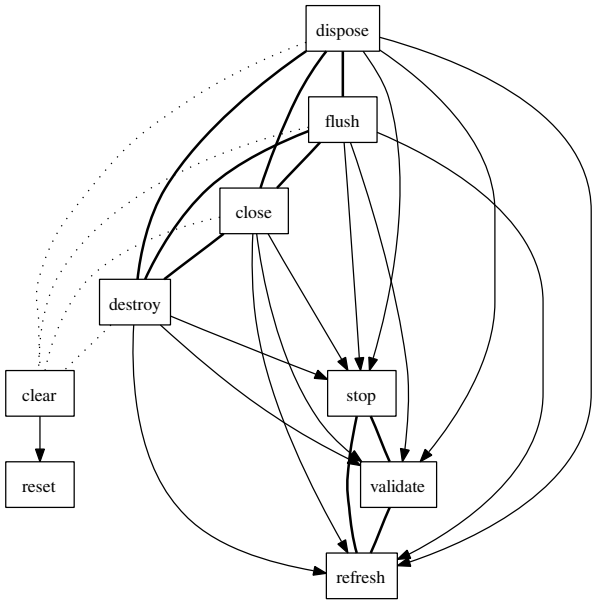


Figure 4: Methods related to termination.

et al. [8] try to approximate the results achieved by Deißenböck and Pizka while avoiding the need for an expert to create the formal model by considering only the syntactic structure of identifiers.

A more thorough analysis of function identifiers is carried out by Caprile and Tonella [1] who investigate the structure of function identifiers in C programs, build a dictionary of identifier fragments ("words"), and propose a grammar that describes the roles of the fragments. The authors also apply concept analysis to perform a classification of the words in the dictionary.

Our work departs significantly from other efforts in the attempt to ground the *semantics* of names in attributes derived from the implementation. In trying to define traceable attributes to correlate to names, we have been influenced by Gil and Maman's work on Micro Patterns [6].

8. Conclusion

We believe that the analysis of semantic relationships between names in computer programs bears many low-hanging fruits, and that in generating *The Programmer's Lexicon*, we have picked but one.

Defining the meaning of names is useful because it leads to greater awareness and might contribute to

more precise use of names. It is easy to envisage a tool, for instance an Eclipse plug-in, that could automatically check whether or not the initial verb of a method name suits the implementation of the method, give warnings when imprecise names are used, and so forth.

More radically, we could detach the action-oriented verbs from the rest of the method names, and raise the verbs to the status of syntax in the language. Then the programmer could write something like `Person find PersonByID`, and have the compiler verify that the implementation is not in conflict with the action specified by the name *find*.

The idea outlined above marks the beginning of a decomposition of the method name into a more operative language, similar to *keyword messages* in Smalltalk [7]. In our present work, we have focused on the verbs that tend to form the beginning of a method name. As Caprile and Tonella [1] have shown, these verbs form part of a structure; a sentence. In the example above, if we were to identify `By` as a special word, the logical next step would be to try to link the identifier fragments `Person` and `ID` to types. The goal would be to generate a full lexicon for all the words that appear in method names, and potentially type names as well.

Since methods tend to invoke other methods, understanding the content of a method body is inherently a recursive problem. In our future work, a key challenge will be to define a suitable model that allows us to define names in terms of other names, much as is the case in natural language.

References

[1] B. Caprile and P. Tonella. Nomen est omen: Analyzing the language of function identifiers. In *Sixth Working Conference on Reverse Engineering (WCRE '99), 6-8 October 1999, Atlanta, Georgia, USA*, pages 112–122. IEEE Computer Society, 1999.

[2] T. M. Cover and J. A. Thomas. *Elements of Information Theory*. Wiley Series in Telecommunications. Wiley, 2nd edition, 2006.

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

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

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Resusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley, 1995.

Phrase	*Meaning*
Always	The attribute value is always 1.
Very often	The name is in the high extreme quantile.
Often	The name is in the high quantile.
Rarely	The name is in the low quantile.
Very seldom	The name is in the low extreme quantile.
Never	The attribute value is always 0.

Table 5: Lexicon terminology.

[6] J. Gil and I. Maman. Micro patterns in Java code. In R. Johnson and R. P. Gabriel, editors, *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.

[7] W. LaLonde. I can read C++ and Java but I can't read Smalltalk. *Journal of Object-Oriented Programming*, pages 40–45, 2000.

[8] D. Lawrie, H. Feild, and D. Binkley. Syntactic identifier conciseness and consistency. In *Sixth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2006), 27-29 September 2006, Philadelphia, Pennsylvania, USA*, pages 139–148. IEEE Computer Society, 2006.

[9] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. The Java Series. Prentice Hall, 2nd edition, 1999.

[10] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, 2006.

Acknowledgements.
We thank Anders Moen Hagalisletto, Thor Kristoffersen and Gerardo Schneider for comments on earlier drafts of this paper.

Appendix
The Lexicon

Introduction.
Below we print *The Programmer's Lexicon*, automatically generated from our analysis of the most common names in the software corpus. In our context, a name is the action-oriented initial part of a Java method name; typically a verb. Like a natural language dictionary, the lexicon does not have to be read in full. Some entries we have found interesting are *check* (throws exceptions), *find* (contains loops) and *equals* (calls methods of the same name, and performs type-checking). Table 5 explains the terminology used in the lexicon.

Lexicon Entries.

accept. Methods named *accept* very seldom read state. Furthermore, they rarely throw exceptions, call methods of the same name, create objects, manipulate state, use local variables, have no parameters, perform type-checking or contain loops. The name *accept* has a precise use. A similar name is *visit*. Generalisations of *accept* are *handle* and *initialize*. Somewhat related names are *set*, *end*, *is* and *insert*.

action. Methods named *action* never call methods of the same name. Furthermore, they very often read state. Finally, they often return void, and rarely throw exceptions, have no parameters or contain loops. The name *action* has a precise use. Similar names are *remove* and *add*.

add. Among the most common method names. Methods named *add* often read state. Similar names are *remove* and *action*.

check. Methods named *check* very often throw exceptions. Furthermore, they often create objects and contain loops, and rarely call methods of the same name. Unfortunately, *check* is an imprecise name for a method.

clear. Methods named *clear* very often have no parameters. Furthermore, they often return void, call methods of the same name and manipulate state, and rarely create objects, use local variables or perform type-checking. A generalisation of *clear* is *reset*. A somewhat related name is *close*.

close. Methods named *close* often return void, call methods of the same name, manipulate state, read state and have no parameters, and rarely create objects or perform type-checking. A generalisation of *close* is *validate*. A somewhat related name is *clear*.

create. Among the most common method names. Methods named *create* very often create objects. Furthermore, they rarely call methods of the same name, read state or contain loops.

do. Methods named *do* often throw exceptions and perform type-checking, and rarely call methods of the same name. Unfortunately, *do* is an imprecise name for a method.

dump. Methods named *dump* never throw exceptions. Furthermore, they very often create objects and use local variables, and very seldom read state. Finally, they often call methods of the same name and contain loops, and rarely manipulate state. The name *dump* has a precise use.

end. Methods named *end* often return void, and rarely create objects, use local variables, read state or contain loops. Generalisations of *end* are *handle* and *initialize*. A specialisation of *end* is *insert*. Somewhat related names are *accept*, *set*, *visit* and *write*.

equals. Methods named *equals* never return void, throw exceptions, create objects, manipulate state or have no parameters. Furthermore, they very often call methods of the same name and perform type-checking. Finally, they often use local variables and read state. The name *equals* has a precise use.

find. Methods named *find* very often use local variables and contain loops. Furthermore, they often perform type-checking, and rarely return void.

generate. Methods named *generate* often create objects, use local variables and contain loops, and rarely call methods of the same name. Unfortunately, *generate* is an imprecise name for a method.

get. The most common method name. Methods named *get* often read state and have no parameters, and rarely return void, call methods of the same name, manipulate state, use local variables or contain loops. A similar name is *has*. Specialisations of *get* are *is* and *size*. A somewhat related name is *hash*.

handle. Methods named *handle* often read state, and rarely call methods of the same name. A similar name is *initialize*. Specialisations of *handle* are *accept*, *set*, *visit*, *end* and *insert*.

has. Methods named *has* often have no parameters, and rarely return void, throw exceptions, create objects, manipulate state, use local variables or perform type-checking. The name *has* has a precise use. A similar name is *get*. Specialisations of *has* are *is* and *size*. A somewhat related name is *hash*.

hash. Methods named *hash* always have no parameters, and never return void, throw exceptions, create objects or perform type-checking. Furthermore, they very often call methods of the same name. Finally, they often read state, and rarely manipulate state or use local variables. The name *hash* has a precise use. Somewhat related names are *has*, *is*, *get* and *size*.

init. Methods named *init* very often manipulate state. Furthermore, they often return void, create objects and have no parameters, and rarely call methods of the same name.

initialize. Methods named *initialize* often return void and manipulate state, and rarely call methods of the same name or read state. A similar name is *handle*. Specialisations of *initialize* are *accept*, *set*, *visit*, *end* and *insert*.

insert. Methods named *insert* often throw exceptions, and rarely create objects, read state, have no parameters or contain loops. Generalisations of *insert* are *handle*, *end* and *initialize*. Somewhat related names are *accept*, *set*, *visit* and *write*.

is. The third most common method name. Methods named *is* often have no parameters, and rarely return void, throw exceptions, call methods of the same name, create objects, manipulate state, use local variables, perform type-checking or contain loops. The name *is* has a precise use. Generalisations of *is* are *has* and *get*. Somewhat related names are *accept*, *visit*, *hash* and *size*.

load. Methods named *load* very often use local variables. Furthermore, they often throw exceptions, create objects, manipulate state, perform type-checking and contain loops. Unfortunately, *load* is an imprecise name for a method.

make. Methods named *make* very often create objects. Furthermore, they rarely return void, throw exceptions, call methods of the same name or contain loops.

new. Methods named *new* never contain loops. Furthermore, they very seldom use local variables. Finally, they often call methods of the same name and create objects, and rarely return void, manipulate state or read state.

next. Methods named *next* very often manipulate state and read state. Furthermore, they often throw exceptions and have no parameters, and rarely return void.

parse. Among the most common method names. Methods named *parse* very often call methods of the same name, read state and perform type-checking. Furthermore, they rarely use local variables. The name *parse* has a precise use.

print. Methods named *print* often call methods of the same name and contain loops, and rarely throw exceptions or manipulate state.

process. Methods named *process* very often use local variables and contain loops. Furthermore, they often throw exceptions, create objects, read state and perform type-checking, and rarely call methods of the same name. Unfortunately, *process* is an imprecise name for a method.

read. Methods named *read* often throw exceptions, call methods of the same name, create objects, manipulate state, use local variables and contain loops. Unfortunately, *read* is an imprecise name for a method.

remove. Among the most common method names. Methods named *remove* often throw exceptions. Similar names are *add* and *action*.

reset. Methods named *reset* very often manipulate state. Furthermore, they often return void and have no parameters, and rarely create objects, use local variables or perform type-checking. A specialisation of *reset* is *clear*.

run. Among the most common method names. Methods named *run* very often read state. Furthermore, they often have no parameters, and rarely call methods of the same name.

set. The second most common method name. Methods named *set* very often manipulate state, and very seldom use local variables or read state. Furthermore, they often return void, and rarely call methods of the same name, create objects, have no parameters, perform type-checking or contain loops. The name *set* has a precise use. Generalisations of *set* are *handle* and *initialize*. Somewhat related names are *accept*, *visit*, *end* and *insert*.

size. Methods named *size* always have no parameters, and never return void, create objects, manipulate state, perform type-checking or contain loops. Furthermore, they very seldom use local variables. Finally, they rarely read state. The name *size* has a precise use. Generalisations of *size* are *has* and *get*. Somewhat related names are *is* and *hash*.

start. Methods named *start* often return void, manipulate state and read state.

to. Among the most common method names. Methods named *to* very often call methods of the same name and create objects. Furthermore, they often have no parameters, and rarely return void, throw exceptions, manipulate state or perform type-checking.

update. Methods named *update* often return void and read state.

validate. Methods named *validate* very often throw exceptions. Furthermore, they often create objects and have no parameters, and rarely manipulate state. A specialisation of *validate* is *close*.

visit. Methods named *visit* rarely throw exceptions, use local variables, read state or have no parameters. A similar name is *accept*. Generalisations of *visit* are *handle* and *initialize*. Somewhat related names are *set*, *end*, *is* and *insert*.

write. Among the most common method names. Methods named *write* often return void and call methods of the same name, and rarely have no parameters. Somewhat related names are *end* and *insert*.