Canonical Method Names for Java

Using Implementation Semantics to Identify Synonymous Verbs

Einar W. Høst[1] and Bjarte M. Østvold[2]

[1] Computas AS
eih@computas.com
[2] Norwegian Computing Center
bjarte@nr.no

Abstract. Programmers rely on the conventional meanings of method names when writing programs. However, these conventional meanings are implicit and vague, leading to various forms of ambiguity. This is problematic since it hurts the readability and maintainability of programs. Java programmers would benefit greatly from a more well-defined vocabulary. Identifying synonyms in the vocabulary of verbs used in method names is a step towards this goal. By rooting the meaning of verbs in the semantics of a large number of methods taken from real-world Java applications, we find that such synonyms can readily be identified. To support our claims, we demonstrate automatic identification of synonym candidates. This could be used as a starting point for a manual canonicalisation process, where redundant verbs are eliminated from the vocabulary.

1 Introduction

Abelson and Sussman [1] contend that "programs must be written for people to read, and only incidentally for machines to execute". This is sound advice backed by the hard reality of economics: maintainability drives the cost of software systems [2], and readability drives the cost of maintenance [3,4]. Studies indicate some factors that influence readability, such as the presence or absence of abbreviations in identifiers [5]. Voices in the industry would have programmers using "good names" [6,7], typically meaning very explicit names. A different approach with the same goal is *spartan programming*[1]. "Geared at achieving the programming equivalent of laconic speech", spartan programming suggests conventions and practical techniques to reduce the complexity of program texts.

We contend that both approaches attempt to fight *ambiguity*. The natural language dimension of program texts, that is, the expressions encoded in the identifiers of the program, is inherently ambiguous. There are no enforced rules regarding the meaning of the identifiers, and hence we get ambiguity in the form of synonyms (several words are used for a single meaning) and polysemes

[1] http://ssdl-wiki.cs.technion.ac.il/wiki/index.php/Spartan_programming

B. Malloy, S. Staab, and M. van den Brand (Eds.): SLE 2010, LNCS 6563, pp. 226–245, 2011.

(a single word has multiple meanings). This ambiguity could be reduced if we managed to establish a more well-defined vocabulary for programmers to use.

We restrict our attention to the first lexical token found in method names. For simplicity, we refer to all these tokens as "verbs", though they need not actually be verbs in English: `to` and `size` are two examples of this. We focus on verbs because they form a central, stable part of the vocabulary of programmers; whereas nouns tend to vary greatly by the domain of the program, the core set of verbs stays more or less intact.

We have shown before [8,9,10] that the meaning of verbs in method names can be modelled by abstracting over the bytecode of the method implementations. This allows us to 1) identify what is typical of implementations that share the same verb, and 2) compare the set of implementations for different verbs. In this paper, we aim at improving the core vocabulary of verbs for Java programmers by identifying potential synonyms that could be unified.

The contributions of this paper are:

- The introduction of *nominal entropy* as a way to measure how "nameable" a method is (Section 3.1).
- A technique to identify methods with "unnameable semantics" based on nominal entropy (Section 4.3).
- A technique to mechanically identify likely instances of code generation in a corpus of methods (Section 4.1).
- A formula to guide the identification of synonymous verbs in method names (Section 3.3).
- A mechanically generated graph showing synonym candidates for the most commonly used verbs in Java (Section 5.1).
- A mechanically generated list of suggestions for canonicalisation of verbs through unsupervised synonym elimination (Section 5.2).

2 Problem Description

To help the readability and learnability of the scripting language PowerShell, Microsoft has defined a standardised set of verbs to use. The verbs and their definitions can be found online[2], and PowerShell programmers are strongly encouraged to follow the conventions. The benefits to readability and learnability are obvious.

By contrast, the set of verbs used in method names in Java has emerged organically, as a mixture of verbs inherited from similar preceding languages, emulation of verbs used in the Java API, and so forth. A similar organic process occurs in natural languages. Steels argues that language "can be viewed as a complex adaptive system that adapts to exploit the available physiological and cognitive resources of its community of users in order to handle their communicative challenges" [11].

[2] `http://msdn.microsoft.com/en-us/library/ms714428%28VS.85%29.aspx`

We have seen before that Java programmers have a fairly homogenous, shared understanding of many of the most prevalent verbs used in Java programs [8]. Yet the organic evolution of conventional verb meaning has some obvious limitations:

- **Redundancy.** There are concepts that evolution has not selected a single verb to represent. This leads to superfluous synonymous verbs for the programmer to learn. Even worse, some programmers may use what are conventional synonyms in subtly different meanings.
- **Coarseness.** It is hard to organically grow verbs with precise meanings. To make sure that a verb is understood, it may be tempting to default to a very general and coarse verb. This results in "bagging" of different meanings into a small set of polysemous verbs.
- **Vagueness.** Evolution in Java has produced some common verbs that are almost devoid of meaning (such as `process` or `handle`), yet are lent a sense of legitimacy simply because they are common and shared among programmers.

Redundancy is the problem of *synonyms*, and can be addressed by identifying verbs with near-identical uses, and choosing a single, canonical verb among them. Coarseness is the problem of *polysemes*, and could be addressed by using data mining to identify common polysemous uses of a verb, and coming up with more precise names for these uses. Vagueness is hard to combat directly, as it is a result of the combination of a lack of a well-defined vocabulary with the programmer's lacking ability or will to create a clear, unambiguous and nameable abstraction. In this paper, we primarily address the problem of redundancy.

3 Analysis of Methods

The meaning of verbs in method names stems from the implementations they represent. That is, the meaning of a verb is simply the collection of observed uses of that verb (Section 3.1). Further, we hold that the verbs become more meaningful when they are consistently used to represent similar implementations. To make it easier to compare method implementations, we employ a coarse-grained semantic model for methods, based on predicates defined on Java bytecode (Section 3.2). We apply entropy considerations to measure both how consistently methods with the same verb are implemented, and how consistently the same implementation is named. We refer to this as semantic and nominal entropy, respectively. These two metrics are combined in a formula that we use to identify synonymous verbs (Section 3.3). Figure 1 presents an overview of the approach.

3.1 Definitions

We define a *method* m as a tuple consisting of three *components*: a unique *fingerprint* u, a *name* n, and a *semantics* s. Intuitively m is an idealised method, a model of a real method in Java bytecode. The unique fingerprints are a technicality that prevents set elements from collapsing into one; hence, a set made

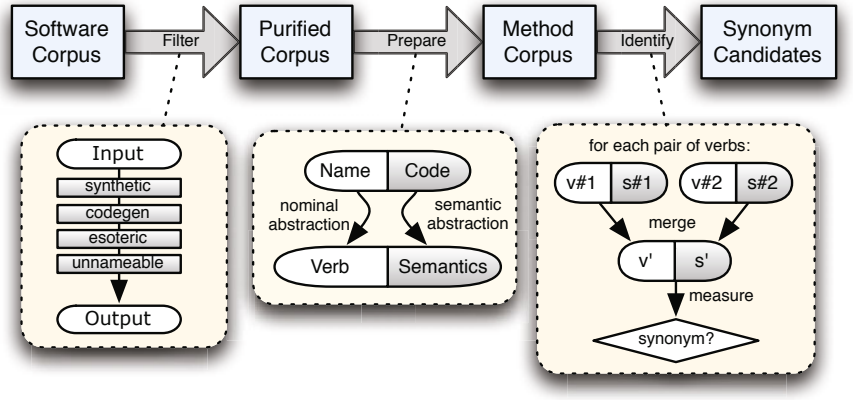


Fig. 1. Overview of the approach

from arbitrary methods $\{m_1, \ldots, m_k\}$ will always have k elements.[3] Often, we elide u from method tuples, writing just $m = (n, s)$.

We need two kinds of languages to reason about methods and their semantics. First, a concrete language where the semantics of a method is simply the string of Java bytecodes in m's implementation. Thus, bytecode is the canonical *concrete language*, denoted as $\mathcal{L}_{\text{Java}}$. For convenience, we define a labelling function f_{MD5} that maps from the bytecode to the MD5 digest of the opcodes in the bytecode. This allows us to easily apply uniform labels to the various implementations.

Second, we need an abstract language consisting of bit-vectors $[b_1, \ldots, b_k]$ where each b_i represents the result of evaluating a logical predicate q_i on a method's implementation. In the context of a concrete method m and its implementation, we refer to the vector $[b_1, \ldots, b_k]$ as *profile* of m. Different choices of predicates $q_1, \ldots, q_k$, leads to different *abstract languages*. Note that with the concrete language there is no limit on the size of a method's semantics; hence there is in principle an unlimited number of semantic objects. With an abstract language there is a fixed number of semantic objects, since s is a k-bit vector for some fixed number k, regardless of the choice of predicates.

A *corpus* $\mathcal{C}$ is a finite set of methods. We use the notation $\mathcal{C}/n$ to denote the set of methods in $\mathcal{C}$ that have name n, but where the semantics generally differs; and similarly $\mathcal{C}/s$ denotes the subcorpus of $\mathcal{C}$ where all methods have semantics s irrespective of their name. $\mathcal{C}/n$ is called a *nominal* corpus, $\mathcal{C}/s$ a *semantic* corpus.

Let x denote either a name component n or a semantics component s of some method. If $x_1, \ldots, x_k$ are all values occurring in $\mathcal{C}$ for a component, then we can view $\mathcal{C}$ as factored into disjoint subcorpora based on these values,

$$\mathcal{C} = \mathcal{C}/x_1 \cup \cdots \cup \mathcal{C}/x_k. \tag{1}$$

[3] The fingerprints models the mechanisms that the run-time system has for identifying distinct callable methods.

Corpus semantics and entropy. We repeat some information-theoretical concepts [12]. A *probability mass function* $p(x)$ is such that *a*) for all $i = 1, \ldots, k$ it holds that $0 \leq p(x_i) \leq 1$; and *b*) $\sum_{i=1}^{k} p(x_i) = 1$. Then $p(x_1), \ldots, p(x_k)$ is a *probability distribution*. From Equation (1) we observe that the following defines a probability mass function:

$$p(\mathcal{C}/x) \stackrel{\text{def}}{=} \frac{|\mathcal{C}/x|}{|\mathcal{C}|}$$

We write p^N for the nominal probability mass function based on name factoring $\mathcal{C}/n$ and Equation (1), and p^S for the semantic version.

We define the semantics $[\![\mathcal{C}]\!]$ of a corpus $\mathcal{C}$ in terms of the distribution defined by p^S:

$$[\![\mathcal{C}]\!] \stackrel{\text{def}}{=} p(\mathcal{C}/s_1) \ldots p^n(\mathcal{C}/s_k)$$

where we assume that $s_1, \ldots, s_k$ are all possible semantic objects in $\mathcal{C}$ as in Equation (1). Of particular interest is the semantics of a nominal corpus; we therefore write $[\![n]\!]$ as a shorthand for $[\![\mathcal{C}/n]\!]$ when $\mathcal{C}$ is obvious from the context. This is what we intuitively refer to as "the meaning of n".

Using the probability mass function, we introduce a notion of entropy for corpora—similar to Shannon entropy [12].

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

where we assume $0 \log_2 0 = 0$. We write $H^N(\mathcal{C})$ for the *nominal entropy* of $\mathcal{C}$, in which case χ denotes the set of all names in $\mathcal{C}$; and $H^S(\mathcal{C})$ for *semantic entropy* of $\mathcal{C}$, where χ denotes the set of all semantics. The entropy $H^S(\mathcal{C})$ is a measure of the semantic diversity of $\mathcal{C}$: High entropy means high diversity, low entropy means few different method implementations. Entropy $H^N(\mathcal{C})$ has the dual interpretation.

Entropy is particularly interesting on subcorpora of $\mathcal{C}$. The nominal entropy of a semantic subcorpus, $H^N(\mathcal{C}/s)$, measures the consistency in the naming methods with profile s in $\mathcal{C}$. The semantic entropy of a nominal subcorpus, $H^S(\mathcal{C}/n)$ measures the consistency in the implementation of name n. The nominal entropy of a nominal subcorpus is not interesting as it is always 0. The same holds for the dual concept. When there can be no confusion about $\mathcal{C}$, we speak of the nominal entropy of a profile s,

$$H^N(s) \stackrel{\text{def}}{=} H^N(\mathcal{C}/s)$$

and similarly for the dual concept $H^S(n)$.

Nominal entropy can be used to compare profiles. A profile with comparatively low nominal entropy indicates an implementation that tends to be consistently named. A profile with comparatively high nominal entropy indicates an ambiguous implementation. An obvious example of the latter is the empty method.

We can also compare the semantic entropy of names. A name with comparatively low semantic entropy implies that methods with that name tend to be implemented using a few, well-understood "cliches". A name with comparatively high semantic entropy implies that programmers cannot agree on what to call such method implementations (or that the semantics are particularly ill-suited at capturing the nature of the name).

We define aggregated entropy of corpus $\mathcal{C}$ as follows.

$$H_{agg}(\mathcal{C}) \stackrel{\text{def}}{=} \frac{\sum_{x \in \chi} |\mathcal{C}/x| H(\mathcal{C}/x)}{|\mathcal{C}|}$$

again leading to nominal and semantic notions of aggregated entropy, $H^N_{agg}(\mathcal{C})$ and $H^S_{agg}(\mathcal{C})$. These notions lets us quantify the overall entropy of subcorpora in $\mathcal{C}$, weighing the entropy of each subcorpus by its size.

Semantic cliches. When a method semantics is frequent in a corpus we call the semantics a semantic cliche, or simply a cliche, for that corpus. When a cliche has many different names we call it an unnameable cliche. Formally, a method semantics s is a *semantic cliche* for a corpus $\mathcal{C}$ if the prevalence of s in $\mathcal{C}$ is above some threshold value ϕ_{cl},

$$\frac{|\mathcal{C}/s|}{|\mathcal{C}|} > \phi_{cl}. \tag{2}$$

Furthermore, s is an *unnameable semantic cliche* if it satisfies the above, and in addition the nominal entropy of corpus $\mathcal{C}/s$ is above some threshold value H^N_{cl}, $H^N(\mathcal{C}/s) > H^N_{cl}$.

3.2 Semantic Model

There are many ways of modelling the semantics of Java methods. For the purpose of comparing method names to implementations, we note one desirable property in particular. While the set of possible method implementations is practically unlimited, the set of different semantics in the model should both be finite and treat implementations that are essentially the same as having the same semantics. This is important, since each $\mathcal{C}/s$ should be large enough so that it is meaningful to speak of consistent or inconsistent naming of the methods in $\mathcal{C}/s$. This ensures that we can judge whether or not methods with semantics s are consistently named.

Some candidates for modelling method semantics are opcode sequences, abstract syntax trees and execution trace sets. However, we find these to be ill suited for our analysis: they do not provide a radical enough abstraction over the implementation. Therefore, we choose to model method semantics using an abstract language of bit vectors, as defined in Section 3.1.

Attributes. The abstract language relies on a set of predicates defined on Java bytecode. We refer to such predicates as *attributes* of the method implementation. Here we select and discuss the attributes we use, which yield a particular abstract language.

Individual attributes cannot distinguish perfectly between verbs. Rather, we expect to see trends when considering the probability that methods in each nominal corpus $\mathcal{C}/n$ satisfy each attribute. Furthermore, we note that 1) there might be names that are practically indistinguishable using bytecode predicates alone, and 2) some names are synonyms, and so *should* be indistinguishable.

Useful attributes. Intuitively, an attribute is *useful* if it helps distinguish between verbs. In Section 3.1, we noted that a verb might influence the probability that the predicate of an attribute is satisfied. Useful attributes have the property that this influence is significant. Attributes can be *broad* or *narrow* in scope. A broad attribute lets us identify larger groups of verbs that are aligned according to the attribute. A narrow attribute lets us identify smaller groups of verbs (sometimes consisting of a single verb). Both can be useful. The goal is to find a collection of attributes that together provides a good distinction between verbs.

Chosen attributes. We hand-craft a list of attributes for the abstract method semantics. An alternative would be to generate all possible simple predicates on bytecode instructions, and provide a selection mechanism to choose the "best" attributes according to some criterion. However, we find it useful to define predicates that involve a combination of bytecodes, for instance to describe control flow or subtleties in object creation. We deem it impractical to attempt a brute force search to find such combinations, and therefore resort to subjective judgement in defining attributes. To ensure a reasonable span of attributes, we pick attributes from the following categories: *method signature*, *object creation*, *data flow*, *control flow*, *exception handling* and *method calls*. The resulting attributes are listed in Table 1.

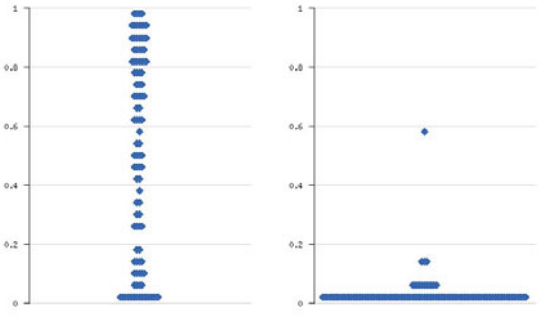
Probability distribution. The probability distribution for an attribute indicates if and how an attribute distinguishes between verbs. To illustrate, Figure 2 shows the probability distribution for two attributes: **Returns void** and **Writes parameter value to field**. Each dot represents the p^v for a given verb v, where v is a "common verb", as defined in Section 4.2. **Returns void** is a broad attribute, that distinguishes well between larger groups of verbs. However, there

Table 1. Attributes

Returns void	Returns field value
Returns boolean	Returns created object
Returns string	Runtime type check
No parameters	Creates custom objects[a]
Reads field	Contains loop
Writes field	Method call
Writes parameter value to field	Returns call result
Throws exceptions	Same verb call
Parameter value passed to method call on field value	

[a] A custom object is an instance of a type not in the `java.*` or `javax.*` namespaces.

are also verbs that are ambiguous with respect to the attribute. By contrast, **Writes parameter value to field** is a narrow attribute. Most verbs have a very low probability for this attribute, but there is a single verb which stands out with a fairly high probability: this verb is `set`, which is rather unsurprising.



(a) *Broad attribute:* **Returns void.**

(b) *Narrow attribute:* **Writes parameter value to field.**

Fig. 2. Probability distribution for some attributes

Critique. We have chosen a set of attributes for the semantic model based on our knowledge of commonly used method verbs in Java and how they are implemented. While all the attributes in the set are *useful* in the sense outlined above, we have no evidence that our set is "optimal" for the task at hand. There are two main problems with this.

First, we might have created an "unbalanced" set of attributes, meaning that we can have too many attributes capturing some kind of behaviour, such as object creation, and too few attributes capturing some other behaviour, such as exception handling. There might even be relevant behaviours that we have omitted altogether.

Second, we can construct many other attributes that could be used to distinguish between names; **Inverted method call**[4] and **Recursive call** are two candidates that we considered but rejected. The former is a narrow attribute that would help characterise `visit` methods, for instance. However, it turns out that `visit` is not ubiquitous enough to be included in our analysis (see Section 4.2); hence the attribute does not help in practise. The latter is simply too rarely satisfied to be very helpful.

The underlying problem is that there is no obvious metric by which to measure the quality of our attribute set. Arguably, the quality — or lack thereof — reveals itself in the results of our analysis.

[4] By "inverted method call", we mean that the calling object is passed as a parameter to the method call.

3.3 Identifying Synonyms

Intuitively, a verb n_1 is redundant if there exists another, more prevalent verb n_2 with *the same meaning*. It is somewhat fuzzy what "the same meaning" means. We define the meaning $[\![n]\!]$ of a verb n as the distribution of profiles in $\mathcal{C}/n$ (see Section 3.1). It is unlikely that the distributions for two verbs will be identical; however, some will be more similar than others. Hence we say that n_1 and n_2 have the same meaning if they are associated with sufficiently similar profile distributions.

We identify synonyms by investigating what happens when we merge the nominal corpora of two verbs. In other words, we attempt to eliminate one of the verbs, and investigate the effects on nominal and semantic entropy. If the effects are beneficial, we have identified a possible synonym.

The effects of synonym elimination. Elimination has two observable effects. First, there is a likely reduction in the aggregated nominal entropy H^N_{agg} of semantic corpora. The reason is that the nominal entropy of an individual semantic corpus is either *unaffected* by the elimination (if the eliminated verb is not used for any of the methods in the corpus), or it is *lowered*. Second, there is a likely increase in the aggregated semantic entropy H^S_{agg} of the nominal corpora — except for the unlikely event that the distribution of profiles is identical for the original corpora $\mathcal{C}/n_1$ and $\mathcal{C}/n_2$. How much H^S_{agg} increases depends on how semantically similar or different the eliminated verb is from the replacement verb. The increase in semantic entropy for the combined nominal corpus will be much lower for synonyms than for non-synonyms.

Optimisation strategy. When identifying synonyms, we must balance the positive effect on nominal entropy with the negative effect on semantic entropy. If we were to ignore the effect on semantic entropy, we would not be considering synonyms at all: simply to combine the two largest nominal corpora would yield the best effect. If we were to ignore the effect on nominal entropy, we would lose sight of the number of methods that are renamed. To combine a very large nominal corpus with a very small one would yield the best effect.

With this in mind, we devise a formula to guide us when identifying synonyms. A naive approach would be to demand that the positive effect on nominal entropy should simply be larger than the negative effect on semantic entropy. From practical experiments, we have found it necessary to emphasise semantic entropy over nominal entropy. That way, we avoid falsely identifying verbs with very large nominal corpora as synonyms. We therefore employ the following optimisation formula, which emphasises balance and avoids extremes, yet is particularly sensitive to increases in semantic entropy:

$$opt(\mathcal{C}) \stackrel{\text{def}}{=} \sqrt{4H^S_{agg}(\mathcal{C})^2 + H^N_{agg}(\mathcal{C})^2}$$

4 Software Corpus

We have gathered a corpus of Java programs of all sizes, from a wide variety of domains. We assume that the corpus is large and varied enough for the code

Table 2. 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		

to be representative of Java programming in general. Table 2 lists the 100 Java applications, frameworks and libraries that constitute our corpus.

We filter the corpus in various ways to "purify" it:

- Omit compiler-generated methods (marked as **synthetic** in the bytecode).
- Omit methods that appear to have been code-generated.
- Omit methods without a common verb-name.
- Omit methods with unnameable semantics.

Table 3. The effects of corpus filtering

Total methods	1.226.611
Non-synthetic	1.090.982
Hand-written	1.050.707
Common-verb name	818.503
Nameable semantics	778.715

The purpose of the filtering is to reduce the amount of noise affecting our analysis. Table 3 presents some numbers indicating the size of the corpus and the impact of each filtering step.

4.1 Source Code Generation

Generation of source code represents a challenge for our analysis, since it can lead to a skewed impression of the semantics of a verb. In our context, the problem is this: a single application may contain a large number of near-identical methods, with identical verb and identical profile. The result is that the nominal corpus corresponding to the verb in question is "flooded" by methods with a specific profile, skewing the semantics of that corpus. Conversely, the semantic corpus corresponding to the profile in question is also "flooded" by methods with a specific verb, giving us a wrong impression of how methods with that profile are named.

To diminish the influence of code generation, we impose limits on the number of method instances contributed by a single application. By comparing the contribution from individual applications to that of all others, we can calculate an *expected* contribution for the application. We compare this with the *actual* contribution, and truncate the contribution if the ratio between the two numbers is unreasonable.

If the actual contribution is above some threshold T, then we truncate it to:

$$\max(T, \min(\frac{|\mathcal{C}_a/v|}{|\mathcal{C}/v|}, L\frac{|\mathcal{C}/v| - |\mathcal{C}_a/v|}{|\mathcal{C}| - |\mathcal{C}_a|}))$$

where L acts as a constraint on how much the contribution may exceed expectations.

Determining T and L is a subjective judgement, since we have no way of identifying false positives or false negatives among the method instances we eliminate. Our goal is to diminish the influence of code generation on our analysis rather than eliminate it. Therefore, we opt to be fairly lax, erring more on the side of false negatives than false positives. In our analysis, $T = 50$ and $L = 25$; that is, if some application contains more than 50 identical methods (n, s), we check that the number of identical methods does not exceed 25 times that of the average application. This nevertheless captures quite a few instances of evident code generation.

Table 4. Vocabularies

Vocabulary	*% Apps*	*Verbs*	*Methods*	*Example verbs*
essential	⟨ 90, 100]	7	50.73%	get, set, create
core	⟨ 75, 90]	21	13.26%	find, equals, parse
extended	⟨ 50, 75]	74	13.92%	handle, match, save
specific	⟨ 25, 50]	220	11.72%	sort, visit, refresh
narrow	⟨ 10, 25]	555	5.95%	render, shift, purge
marginal	⟨ 0, 10]	5722	4.43%	squeeze, unhook, animate

4.2 Common Verbs

Some verbs are common, such as `get` and `set`, whereas others are esoteric, such as `unproxy` and `scavenge`. In this paper, we focus on the former and ignore the latter. There are several possible interpretations of *common*; two obvious candidates are *ubiquity* (percentage of applications) and *volume* (number of methods).

We choose ubiquity as our interpretation of *common*. Rudimentary grouping of verbs according to ubiquity is shown in Table 4. Since we are interested in the shared vocabulary of programmers, we restrict our analysis to the top three groups: *essential*, *core* and *extended*. The 102 verbs in these three groups cover nearly 77% of all methods (after filtering of generated code). Figure 3 shows a "word cloud" visualisation[5] of the common verbs.

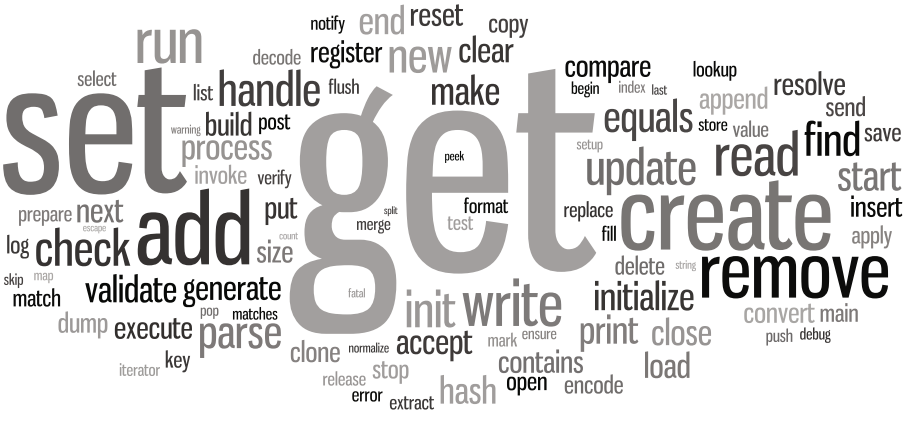


Fig. 3. The 102 most common verbs

4.3 Unnameable Cliches

Unnameable cliches, that is, method implementations that are common, yet inherently ambiguous, constitute noise for our analysis. We aim to reduce the

[5] Generated by Wordle.net.

impact of this noise by omitting methods whose implementations are unnameable cliches. The rationale is that the semantics of each verb will be more distinct without the noise, making it easier to compare and contrast the verbs.

In some cases, an implementation cliche may appear to be unnameable without being inherently ambiguous: rather, no generally accepted name for it has emerged. By applying canonicalisation through synonym elimination, the naming ambiguity can be reduced to normal levels. We must therefore distinguish between cliches that are *seemingly* and *genuinely* unnameable.

To identify implementation cliches, we use the concrete language $\mathcal{L}_{\text{Java}}$ (see Section 3.1). Table 5 shows unnameable cliches identified using Equation (2), with $\phi_{cl} = 500$ and $H^N_{cl} = 1.75$. We also include a reverse engineered example in a stylized Java source code-like syntax for each cliche.

To label the method implementations we apply f_{MD5}, which yields the MD5 digest of the opcodes for each implementation. Note that we include only the opcodes in the digest. We omit the operands to avoid distinguishing between implementations based on constants, text strings, the names of types and methods, and so forth. Hence f_{MD5} does abstract over the implementation somewhat. As a consequence, we cannot distinguish between, say, `this.`m`(`p`)` and p`.`m`(this)`: these are considered instances of the same cliche. Also, some cliches may yield the same example, since there are opcode sequences that cannot be distinguished when written as stylized source code.

Most of the cliches in Table 5 seem genuinely unnameable. Unsurprisingly, variations over delegation to other methods dominate. We cannot reasonably

Table 5. Semantic cliches with unstable naming

Cliche	*# Methods*	H^N	*Retain*	*Top names*
{ super.m(); }	539	3.50		remove [10.6%], set [8.7%], insert [6.5%]
{ }	14566	3.40		set [18.1%], initialize [8.4%], end [7.8%]
{ this.m(); }	794	3.22		set [18.5%], close [7.2%], do [6.2%]
{ this.f.m(); }	2007	2.94		clear [20.7%], close [13.2%], run [11.7%]
{ return p; }	532	2.94		get [34.4%], convert [7.1%], create [4.7%]
{ super.m(p); }	742	2.69		set [32.2%], end [12.0%], add [9.4%]
{ throw new E(); }	3511	2.68		get [25.4%], remove [17.2%], set [14.8%]
{ this.f.m(); }	900	2.65		clear [28.2%], remove [16.1%], close [9.9%]
{ throw new E(s); }	5666	2.59		get [25.9%], set [22.3%], create [10.7%]
{ this.f.m(p); }	1062	2.48		set [39.2%], add [14.8%], remove [12.0%]
{ this.m(p); }	1476	2.45		set [24.4%], end [21.7%], add [14.2%]
{ return this.f.m(p); }	954	2.38		contains [25.9%], is [20.8%], equals [11.1%]
{ this.f.m(p_1, p_2); }	522	2.34		set [33.0%], add [17.2%], remove [13.0%]
{ return this.f.m(p); }	929	2.14		contains [28.3%], is [25.0%], get [11.1%]
{ return this.m(p); }	618	2.14		get [52.8%], post [8.4%], create [6.3%]
{ this.f = true; }	631	2.08	✓	set [48.5%], mark [12.8%], start [6.7%]
{ C.m(this.f); }	544	1.96		run [46.9%], handle [14.3%], insert [9.9%]
{ this.f.m(p); }	3906	1.92		set [36.8%], add [29.7%], remove [16.8%]
{ return new C(this); }	1540	1.87	✓	create [34.6%], get [25.7%], new [11.9%]
{ return this.m(); }	520	1.83		get [45.0%], is [20.0%], has [12.5%]
{ return false; }	6322	1.83	✓	is [52.8%], get [20.1%], has [7.3%]

provide a name for such methods without considering the names of the methods being delegated to. There are also some examples of "unimplemented" methods; for instance `{ throw new E(); }` or the empty method `{ }`. We believe that in many cases, the presence of these methods will be required by the compiler (for instance to satisfy some interface), but in practice, they will never be invoked.

Table 5 also contains three cliches that we deem only seemingly unnameable. This is based on a subjective judgement that they could be relatively consistently named, given a more well-defined vocabulary. These have been marked as being *retained*, meaning they are included in the analysis. The others are omitted.

5 Addressing Synonyms

To address the problem of synonyms, we employ the formula *opt* from Section 3.3. We use *opt* to mechanically identify likely synonyms in the corpus described in Section 4, and then to attempt unsupervised elimination of synonyms.

5.1 Identifying Synonyms

Compared to each of the common verbs in the corpus, the other verbs will range from synonyms or "semantic siblings" to the opposite or unrelated. To find the verbs that are semantically most similar to each verb, we calculate the value for *opt* when merging the nominal corpus of each verb with the nominal corpus of each of the other verbs. The verbs that yield the lowest value for *opt* are considered synonym candidates.

It is more likely that two verbs are genuine synonyms if they reciprocally hold each other to be synonym candidates. When we identify such pairs of synonym candidates, we find that clusters emerge among the verbs, as shown in Figure 4.

Several of the clusters could be labelled, for instance as *questions*, *initialisers*, *factories*, *runners*, *checkers* and *terminators*. This suggests that these clusters have a "topic". It does not imply that all the verbs in each cluster could be replaced by a single verb, however. For instance, note that in the *factory* cluster, `create` and `make` are indicated as synonym candidates, as are `create` and `new`, but `new` and `make` are not. An explanation could be that `create` has a broader use than `new` and `make`.

We also see that there are two large clusters that appear to have more than one topic. We offer two possible explanations. First, polysemous verbs will tie together otherwise unrelated topics (see Section 2). In the largest cluster, for instance, we find a mix of verbs associated with I/O and verbs that handle collections. In this case, `append` is an example of a polysemous verb used in both contexts. Second, we may lack attributes to distinguish appropriately between the verbs.

5.2 Eliminating Synonyms

To eliminate synonyms, we iterate over the collection of verbs. We greedily select the elimination that yields the best immediate benefit for *opt* in each iteration.

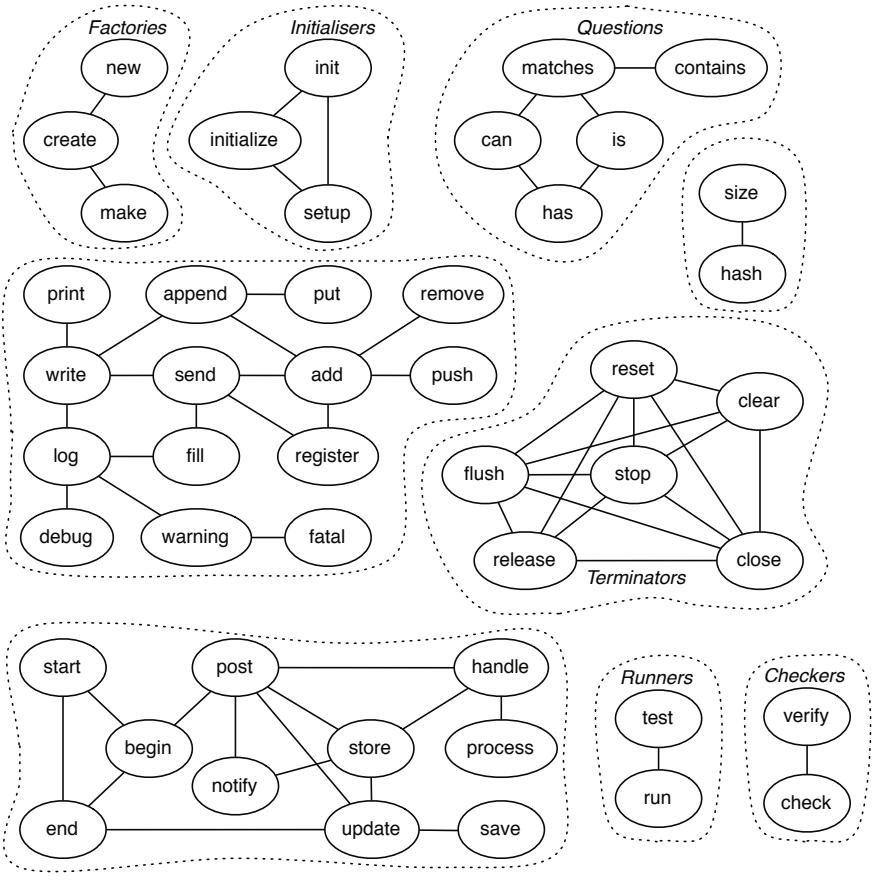


Fig. 4. Clusters of synonym candidates. Clusters with a single topic are labelled.

Table 6. Mechanical elimination of synonyms

Run	*Canonical verb (cv)*	*Old verbs*	$\lvert\mathcal{C}/cv\rvert$	*Sum*	ΔH^S_{agg}	ΔH^N_{agg}	Δopt
1	is	has+is	49041	6820+42221	0.00269	-0.02270	-0.01152
2	is	can+is	51649	2608+49041	0.00178	-0.01148	-0.00409
3	add	remove+add	43241	16172+27069	0.00667	-0.03004	-0.00237
4	init	initialize+init	11026	3568+7458	0.00149	-0.00743	-0.00126
5	close	stop+close	5025	1810+3215	0.00074	-0.00348	-0.00040
6	create	make+create	38140	4940+33200	0.00363	-0.01525	-0.00021
7	close	flush+close	5936	911+5025	0.00061	-0.00266	-0.00014
8	reset	clear+reset	5849	2901+2948	0.00100	-0.00421	-0.00007
9	write	log+write	13659	1775+11884	0.00131	-0.00547	-0.00004

Table 7. Manual elimination of synonyms

Canonical verb (cv)	*Old verbs*	$\lvert\mathcal{C}/cv\rvert$	*Sum*	ΔH^S_{agg}	ΔH^N_{agg}	Δopt
clone	clone+copy	4732	2595+2137	0.00271	-0.00147	0.00979
execute	execute+invoke	4947	2997+1950	0.00197	-0.00229	0.00589
verify	check+verify	8550	7440+1110	0.00126	-0.00298	0.00223
stop	stop+end	4814	1810+3004	0.00126	-0.00283	0.00242
write	write+log+dump	15987	11884+1775+2328	0.00420	-0.01109	0.00635
start	start+begin	5485	4735+750	0.00081	-0.00200	0.00135
init	init+initialize	11026	7458+3568	0.00149	-0.00743	-0.00126
error	error+fatal	1531	1116+415	0.00027	-0.00088	0.00023
create	create+new+make	45565	33200+7425+4940	0.00901	-0.03588	0.00152

We assume that beneficial eliminations will occur eventually, and that the order of eliminations is not important. We only label a synonym candidate as "genuine" if the value for *opt* decreases; the iteration stops when no more genuine candidates can be found. For comparison, we also perform manual elimination of synonyms, based on a hand-crafted list of synonym candidates.

The results of mechanical synonym elimination are shown in Table 6. Note that the input to the elimination algorithm is the output given by the preceding run of the algorithm. For the first run, the input is the original "purified" corpus described in Section 4, whereas for the second, the verb `has` has been eliminated, and the original nominal corpora for `has` and `is` have been merged.

The elimination of `has` is interesting: it is considered the most beneficial elimination by *opt*, yet as Java programmers, we would hesitate to eliminate it. The subtle differences in meaning between all "boolean queries" (`is`, `has`, `can`, `supports` and so forth) are hard to discern at the implementation level. Indeed, we would often accept method names with different verbs for the same implementation: `hasChildren` and `isParent` could be equally valid names. This kind of nominal flexibility is arguably beneficial for the readability of code.

It is easier to see that either `init` or `initialize` should be eliminated: there is no reason for the duplication. Eliminating `make` and using `create` as a canonical verb for factory methods also seems reasonable. Similarly, the suggestion to use `write` instead of `log` is understandable — however, one could argue that `log` is useful because it is more precise than the generic `write`.

There seems to be quite a few verbs for "termination code"; some of these verbs might be redundant. The unsupervised elimination process identifies `flush`, `stop` and `close` as candidates for synonym elimination. However, we find it unacceptable: certainly, `flush` and `close` cannot always be used interchangeably. In our coarse-grained semantic model, we lack the "semantic clues" to distinguish between these related, yet distinct verbs.

The suggestion to combine `add` and `remove` is also problematic, again showing that the approach has some limitations. Both `add` and `remove` typically involve collections of items, perhaps including iteration (which is captured by the **Contains loop** attribute). The crucial distinction between the two operations will often be hidden inside a call to a method in the Java API. Even if we were to

observe the actual adding or removing of an item, this might involve incrementing or decrementing a counter, which is not captured by our model.

Table 7 shows the result of the manual elimination of synonyms. We note that only the elimination of `initialize` yields a decreased value for *opt* — apparently, we are not very good at manual synonym identification! However, it may be that the requirement that *opt* should decrease is too strict. Indeed, we find that many of our candidates are present in the clusters shown in Figure 4. This indicates that there is no deep conflict between our suggestions and the underlying data.

5.3 Canonicalisation

Overall, we note that our approach succeeds in finding relevant candidates for synonym elimination. However, it is also clear that the elimination must be supervised by a programmer. We therefore suggest using Figure 4 as a starting point for manual canonicalisation of verbs in method names. Canonicalisation should entail both eliminating synonyms and providing a precise definition, rationale and use cases for each verb.

6 Related Work

Gil and Maman [13] introduce the notion of machine-traceable patterns, in order to identify so-called micro patterns; machine-traceable implementation patterns at the class level. When we model the semantics of method implementations using hand-crafted bytecode predicates, we could in principle discern "nano patterns" at the method implementation level. According to Gamma et al. [14], however, a pattern has four essential elements: name, problem, solution and consequences. Though we do identify some very commonly used implementation cliches, we do not attempt to interpret and structure these cliches. Still, Singer et al. [15] present their own expanded set of bytecode predicates under the label "fundamental nano patterns", where the term "pattern" must be understood in a broader, more colloquial sense.

Collberg et al. [16] present a large set of low-level statistics from a corpus of Java applications, similar in size to ours. Most interesting to us are the statistics showing k-grams of opcodes, highlighting the most commonly found opcode sequences. This is similar to the implementation cliches we find in our work. Unfortunately, the k-grams are not considered as logical entities, so a common 2-gram will often appear as part of a common 3-gram as well.

Similar in spirit to our work, Singer and Kirkham [17] find a correlation between certain commonly used type name suffixes and some of Gil and Maman's micro patterns. Pollock et al. [18] propose using "natural language program analysis", where natural language clues found in comments and identifiers are used to augment and guide program analyses. Tools for program navigation and aspect mining have been developed [19,20] based on this idea. Ma et al. [21] exploit the fact that programmers usually choose appropriate names in their code to guide searches for software artefacts.

The quality of identifiers is widely recognised as important. Deißenböck and Pizka [22] seek to formalise two quality metrics, *conciseness* and *consistency*, based on a bijective mapping between identifers and concepts. Unfortunately, the mapping must be constructed by a human expert. Lawrie et al. [23] seek to overcome this problem by deriving syntactic rules for conciseness and consistency from the identifiers themselves. This makes the approach much more applicable, but introduces the potential for false positives and negatives.

7 Conclusion and Further Work

The ambiguous vocabulary of verbs used in method names makes Java programs less readable than they could be. We have identified *redundancy*, *coarseness* and *vagueness* as the problems to address. In this paper, we focussed on *redundancy*, where more than one verb is used in the same meaning. We looked at the identification and elimination of synonymous verbs as a means towards this goal.

We found that we were indeed able to identify reasonable synonym candidates for many verbs. To select the genuine synonyms among the candidates without human supervision is more problematic. The abstract semantics we use for method implementations is sometimes insufficient to capture important nuances between verbs. A more sophisticated model that takes into account invoked methods, either semantically (by interprocedural analysis of bytecode) or nominally (by noting the names of the invoked methods) might overcome some of these problems. Realistically, however, the perspective of a programmer will probably still be needed. A more fruitful way forward may be to use the identified synonym candidates as a starting point for a manual process where a canonical set of verbs is given precise definitions, and the rest are discouraged from use.

Addressing the problem of *coarseness* is a natural counterpart to the topic of this paper. Coarseness manifests itself in polysemous verbs, that is, verbs that have more than a single meaning. Polysemous verbs could be addressed by investigating the semantics of the methods that constitute a nominal corpus $\mathcal{C}/n$. The intuition is that polysemous uses of n will reveal itself as clusters of semantically similar methods. Standard data clustering techniques could be applied to identify such polysemous clusters. If a nominal corpus were found to contain polysemous clusters, we could investigate the effect of renaming the methods in one of the clusters. This would entail splitting the original nominal corpus $\mathcal{C}/n$ in two, $\mathcal{C}/n$ and $\mathcal{C}/n'$. The effect of splitting the corpus could be measured, for instance by applying the formula *opt* from Section 3.3.

References

1. Abelson, H., Sussman, G.J.: Structure and Interpretation of Computer Programs, 2nd edn. MIT Electrical Engineering and Computer Science. MIT Press, Cambridge (1996)
2. Eierman, M.A., Dishaw, M.T.: 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)

3. Collar, E., Valerdi, R.: Role of software readability on software development cost. In: Proceedings of the 21st Forum on COCOMO and Software Cost Modeling, Herndon, VA (October 2006)
4. von Mayrhauser, A., Vans, A.M.: Program comprehension during software maintenance and evolution. Computer 28(8), 44–55 (1995)
5. Lawrie, D., Morrell, C., Feild, H., Binkley, D.: Effective identifier names for comprehension and memory. ISSE 3(4), 303–318 (2007)
6. Martin, R.C.: Clean Code. Prentice-Hall, Englewood Cliffs (2008)
7. Beck, K.: Implementation Patterns. Addison-Wesley Professional, Reading (2007)
8. Høst, E.W., Østvold, B.M.: The programmer's lexicon, volume I: The verbs. In: Proceedings of the 7th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2007), pp. 193–202. IEEE Computer Society, Los Alamitos (2007)
9. Høst, E.W., Østvold, B.M.: The java programmer's phrase book. In: Gašević, D., Lämmel, R., Van Wyk, E. (eds.) SLE 2008. LNCS, vol. 5452, pp. 322–341. Springer, Heidelberg (2009)
10. Høst, E.W., Østvold, B.M.: Debugging method names. In: Drossopoulou, S. (ed.) ECOOP 2009. LNCS, vol. 5653, pp. 294–317. Springer, Heidelberg (2009)
11. Steels, L.: The recruitment theory of language origins. In: Lyon, C., Nehaniv, C.L., Cangelosi, A. (eds.) Emergence of Language and Communication, pp. 129–151. Springer, Heidelberg (2007)
12. Cover, T.M., Thomas, J.A.: Elements of Information Theory, 2nd edn. Wiley Series in Telecommunications. Wiley, Chichester (2006)
13. Gil, J., Maman, I.: Micro patterns in Java code. In: Proceedings of the 20th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2005), pp. 97–116. ACM, New York (2005)
14. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns. Addison-Wesley, Boston (1995)
15. Singer, J., Brown, G., Lujan, M., Pocock, A., Yiapanis, P.: Fundamental nanopatterns to characterize and classify Java methods. In: Proceedings of the 9th Workshop on Language Descriptions, Tools and Applications (LDTA 2009), pp. 204–218 (2009)
16. Collberg, C., Myles, G., Stepp, M.: An empirical study of Java bytecode programs. Software Practice and Experience 37(6), 581–641 (2007)
17. Singer, J., Kirkham, C.: Exploiting the correspondence between micro patterns and class names. In: Proceedings of the 8th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2008), pp. 67–76. IEEE Computer Society, Los Alamitos (2008)
18. Pollock, L.L., Vijay-Shanker, K., Shepherd, D., Hill, E., Fry, Z.P., Maloor, K.: Introducing natural language program analysis. In: Das, M., Grossman, D. (eds.) Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE 2007), pp. 15–16. ACM, New York (2007)
19. Shepherd, D., Pollock, L.L., Vijay-Shanker, K.: Towards supporting on-demand virtual remodularization using program graphs. In: Filman, R.E. (ed.) Proceedings of the 5th International Conference on Aspect-Oriented Software Development (AOSD 2006), pp. 3–14. ACM, New York (2006)
20. Shepherd, D., Fry, Z.P., Hill, E., Pollock, L., Vijay-Shanker, K.: 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), pp. 212–224. ACM, New York (2007)

21. Ma, H., Amor, R., Tempero, E.D.: Indexing the Java API using source code. In: Proceedings of the 19th Australian Software Engineering Conference (ASWEC 2008), pp. 451–460. IEEE Computer Society, Los Alamitos (2008)
22. Deißenböck, F., Pizka, M.: Concise and consistent naming. In: Proceedings of the 13th IEEE International Workshop on Program Comprehension (IWPC 2005), pp. 97–106. IEEE Computer Society, Los Alamitos (2005)
23. Lawrie, D., Feild, H., Binkley, D.: Syntactic identifier conciseness and consistency. In: 6th IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2006), pp. 139–148. IEEE Computer Society, Los Alamitos (2006)