The Java Programmer's Phrase Book

Einar W. Høst and Bjarte M. Østvold

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

Abstract. Method names in Java are natural language phrases describing behaviour, encoded to make them easy for machines to parse. Programmers rely on the meaning encoded in method names to understand code. We know little about the language used in this encoding, its rules and structure, leaving the programmer without guidance in expressing her intent. Yet the meaning of the method names — or phrases — is readily available in the body of the methods they name. By correlating names and implementations, we can figure out the meaning of the original phrases, and uncover the rules of the phrase language as well. In this paper, we present an automatically generated proof-of-concept phrase book for Java, based on a large software corpus. The phrase book captures both the grammatical structure and the meaning of method phrases as commonly used by Java programmers.

1 Introduction

Method identifiers play three roles in most programming languages. The first is a technical one: method identifiers are *unique labels* within a class; strings of characters that act as links, allowing us to unambiguously identify a piece of code. If we want to invoke that piece of code, we refer to the label. The second role is mnemonic. While we could, in theory, choose arbitrary labels for our methods, this would be cumbersome when trying to remember the correct label for the method we want to invoke. Hence methods are typically given labels that humans can remember, leading us to refer to method identifiers also as method *names*. Finally, method identifiers play a semantic role. Not only do we want labels we can remember; we want them to express meaning or intent. This allows us to recall what the method actually does. Unfortunately, we lack an established term for this role — identifiers are not just names. Rather, they are structured expressions of intent, composed of one or more fragments. Indeed they are *method phrases*, utterances in natural language, describing the behaviour of methods.

Consider, for instance, the following example, defining the Java method `findElementByID`:

```
 Element findElementByID(String id) {
   for (Element e : this.elements) {
     if (e.getID().equals(id)) {
       return e;
     }
   }
   return null;
 }
```

We immediately observe that the form of the method phrase is somewhat warped and mangled due to the hostile hosting environment. Since the phrase must double as a unique label for the method, and to simplify parsing, the phrase must be represented by a continuous strings of characters. But it is nevertheless a phrase, and we have no problems identifying it as such. In a more friendly environment, we would unmangle the phrase and simply write *Find element by ID*. For instance, in the programming language Subtext[1], the roles as links and names are completely decoupled: the names are mere comments for the links. This leaves the programmer with much greater flexibility when naming methods. Edwards argues that “names are too rich in meaning to waste on talking to compilers” [1].

Looking at the implementation, we see that there are several aspects that conspire to match the expectations given us by the phrase: the method returns a reference to an object, accepts a parameter, contains a loop, and has two return points. We posit that all meaningful method phrases can similarly be described, simply by noting the distinguishing aspects of their implementations. Our goal is to automatically generate such descriptions for the most common method phrases in Java.

Since the phrase and the implementation of a method should be in harmony, we cannot arbitrarily change one without considering to change the other. We want the phrase to remain a correct abstract description of the implementation of the method, otherwise the programmer is lying! Therefore, if the implementation is changed, the phrase should ideally be changed to describe the new behaviour[2]. Conversely, if the phrase is changed, care should be taken to ensure that the implementation fulfills the promise of the new phrase. Unfortunately, programmers have no guidance besides their own intuition and experience to help make sure that their programs are truthful in this sense. In particular, programmers lack the ability to assess the quality of method phrases with regards to suitability, accuracy and consistency.

Realizing that method names are really phrases, that is, expressions in natural language, allows us some philosophical insight into the relationship between the method name and the method body or implementation. Frege distinguishes between the *sign* — name, or combination of words —, the *reference* — the object to which the sign refers — and the *sense* — our collective understanding of the reference [2]. Note that the sense is distinct from what Frege calls the *idea*, which is the individual understanding of the reference. Depending on the

[1] `http://subtextual.org`

[2] However, since phrases act as links, this is not always practical: changing phrases in public APIs may break client code.

insight of the individual, the idea (of which there are many) may be in various degrees of harmony or conflict with the sense (of which there is only one).

In this light, the creation of a method is a way of expression where the programmer provides both the sign (the method name) and a manifestation of the idea (the implementation). The tension between the individual idea and the sense is what motivates our work: clearly it would be valuable to assist the programmer in minimizing that tension. Understanding the language of method phrases is a first step towards providing non-trivial assistance to programmers in the task of naming. In the future, we will implement such assistance in a tool.

We have previously shown how to create a semantics which captures our common interpretation, or sense, of the *action verbs* in method names [3]. Building on this work, we use an augmented model for the semantics, and expand from investigating verbs to full method names, understood as natural language phrases.

The main contributions of this paper are as follows:

– A perspective on programming that treats method names formally as expressions in a restricted natural language.
– The identification of a restricted natural language, *Programmer English*, used by Java programmers when writing the names of methods (Section 2.1).
– An approach to encoding the semantics of methods (Section 3.2), expanding on our previous work.
– An algorithm for creating a relevant and useful phrase book for Java programmers (Section 4.2).
– A proof-of-concept phrase book for Java that shows the potential and practicality of our approach (see Section 5 for excerpts).

2 Conceptual Overview

Our goal is to describe the meaning and structure of method names as found in "the real world" of Java programming, and present the findings in a *phrase book* for programmers. Our approach is to compare method names with method implementations in a large number of Java applications. In doing so, we are inspired by Wittgenstein, who claimed that "the meaning of a word is its use in the language" [4]. In other words, the meaning of natural language expressions is established by pragmatically considering the contexts in which the expressions are used.

2.1 Programmer English

Method names in Java are phrases written in a natural language that closely resembles English. However, the language has important distinguishing characteristics stemming from the context in which it is used, affecting both the grammar and the vocabulary of the language.

The legacy of short names lingering from the days before support for automatic name completion still influences programmers. This results in abbreviations and degenerate names with little grammatical structure. While increasing focus on readability might have improved the situation somewhat, Java is still haunted by this culture. A recent example is the `name` method defined for *enums*, a language feature introduced in Java 5.0. A more explicit name would be `getName`.

Futhermore, the vocabulary is filled with general computing terms, technology acronyms, well-known abbreviations, generic programming terms and special object-oriented terms. In addition, the vocabulary of any given application is extended with domain terms, similar to the use of foreign words in regular English. This vocabulary is mostly understandable to programmers, but largely incomprehensible to the English-speaking layman.

We therefore use the term *Programmer English* to refer to the special dialect of English found in Java method names. Of course, Programmer English is really *Java Programmer English*, and other "Programmer Englishes" exist which might exhibit quite different characteristics. For instance, *Haskell Programmer English* is likely to be radically different, whereas *Ruby Programmer English* probably shares some traits with Java Programmer English, and *C# Programmer English* is likely to be near-identical.

2.2 Requirements for The Phrase Book

The main requirements for a phrase book is that it be *relevant* and *useful*. For the phrase book to be relevant, it must stem from "the trenches" of programming. In other words, it must be based on real-world data (i.e. programs), and be representative of how typical Java programmers express themselves.

The usefulness requirement is somewhat more subtle: what does it mean to be useful? Certainly, the phrase book should have a certain amount of *content*, and yet be *wieldy*, easy to handle for the reader. Hence, we want to be able to adjust the number of phrases included in the phrase book. In addition, each phrase must be useful in itself. We propose the following three requirements to ensure the usefulness of phrases: 1) each phrase must have a description that matches actual usage, 2) each phrase must have a well-understood semantics, and 3) each phrase must be widely applicable. These are requirements for *validity*, *precision* and *ubiquity*, respectively.

Since each Java application has its own specialized vocabulary, we must be able to abstract away domain-specific words. The phrase book should therefore contain both concrete phrases such as **get-last-element** and abstract ones such as **find-[noun]**. We prefer concrete phrases since they are more directly applicable, but need abstract phrases to fill out the picture.

We also decide that the phrase book should be organized hierarchically, as a tree. That way, the phrase book directly reflects and highlights the grammatical structure of the phrases themselves. This also makes the phrase book easier to browse. The phrases should therefore be organized as refinements of each other. Since a phrase represents a set of methods, its refinements are a partitioning of the set. Note that the partitioning is syntax-driven: we cannot choose freely which methods to group together. At any given step, we can only choose refined phrases supported by the grammar implicitly defined by the corpus.

2.3 Approach

Figure 1 provides an overview of our approach. The analysis consists of two major phases: data preparation and phrase book generation.

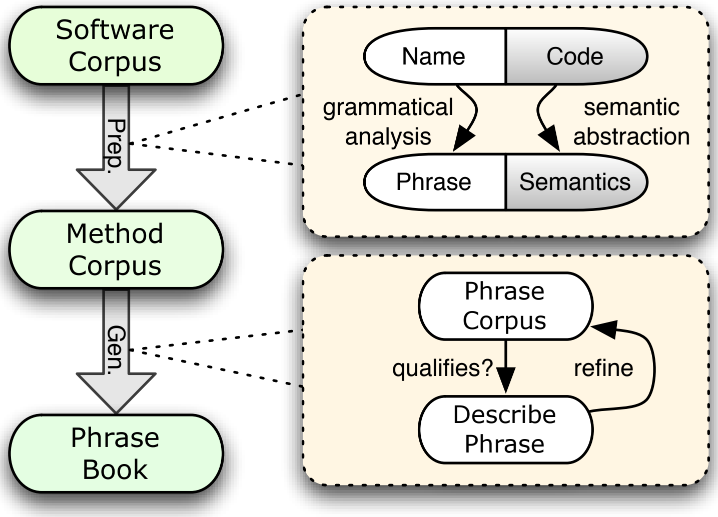


Fig. 1. Overview of the approach.

In the preparation phase, we transform our corpus of Java applications into an idealized corpus of methods. Each Java method is subject to two parallel analyses. On one hand, we analyze the grammatical structure of the method name. This analysis involves decomposing the name into individual words and the part-of-speech tagging of those words. This allows us to abstract over the method names and create phrases consisting of both concrete words and abstract categories of words. On the other hand, we analyze the bytecode instructions of the implementations, and derive an abstract semantics. We can then investigate the semantics of the methods that share the same phrase, and use this to characterize the phrase.

This is exactly what happens in the generation phase. We apply our recursive phrase book generation algorithm on the corpus of methods. The algorithm works by considering the semantics of gradually more refined phrases. The semantics of a phrase is determined by the semantics of the individual methods with names that match the phrase. If a phrase is found to be useful, it receives a description in the phrase book. We generate a description by considering how the phrase semantics compares to that of others. We then attempt to refine the phrase further. When a phrase is found to be not useful, we terminate the algorithm for that branch of the tree.

2.4 Definitions

We need some formal definitions to be used in the analysis of methods (Sect. 3) and when engineering the phrase book (Sect. 4). First, we define a set $\mathcal{A}$ of attributes that each highlight some aspect of a method implementation. An *attribute* $a \in \mathcal{A}$ can be evaluated to a binary value $b \in \{0,1\}$, denoting absence or presence. A *method* m has three features: a unique *fingerprint* u, a *name* n, and a list of *values* $b_1, \ldots, b_n$ for each $a \in A$. These values are the *semantics* of the method. Unique fingerprints ensure that a set made from arbitrary methods $m_1, \ldots, m_k$ always has k elements. The name n consists of one or more *fragments* f. Each fragment is annotated with a *tag* t.

A *phrase* p is an abstraction over a method name, consisting of one or more *parts*. A part may be a fragment, a tag or a special wildcard symbol. The wildcard symbol, written $*$, may only appear as the last part of a phrase. A phrase that consists solely of fragments is *concrete*; all other phrases are *abstract*. A concrete phrase, then, is the same as a method name.

A phrase *captures* a name if each individual part of the phrase captures each fragment of the name. A fragment part captures a fragment if they are equal. A tag part captures a fragment if it is equal to the fragment's tag. A wildcard part captures any remaining fragments in a name. A concrete phrase can only capture a single name, whereas an abstract phrase can capture multiple names. For instance, the phrase **[verb]-valid-*** captures names like **is-valid**, **has-valid-signature** and so forth. The actual set of captured names is determined by the corpus.

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

$$\xi_a(\mathcal{C}) \overset{\text{def}}{=} \frac{\sum_{m \in \mathcal{C}} b_a(m)}{|\mathcal{C}|},$$

where $b_a(m)$ is the binary value for the attribute a of method m. The semantics of a corpus is defined as the list of frequency values for all $a \in \mathcal{A}$, $[\xi_{a_1}(\mathcal{C}), \ldots, \xi_{a_m}(\mathcal{C})]$. We write $[\![p]\!]$ for the semantics of a phrase, and define it as the semantics of the corresponding phrase corpus.

If two methods m, m' have the same values for all attributes we say that they are *attribute-value identical*, denoted $m \simeq m'$. Using relation $\simeq$ we can divide a corpus $\mathcal{C}$ into a set of equivalence classes $\mathrm{EC}(\mathcal{C}) = [m_1]_\mathcal{C}, \ldots, [m_k]_\mathcal{C}$, where $[m]_\mathcal{C}$ is defined as:

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

We simplify the notation to $[m]$ when there can be no confusion about the interpretation of $\mathcal{C}$. Now we apply some information-theoretical concepts related to entropy [5]. Let the probability mass function $p([m])$ of corpus $\mathcal{C}$ be defined as:

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

We then define the *entropy* of corpus $\mathcal{C}$ as:

$$H(\mathcal{C}) \overset{\text{def}}{=} H\big(p([m_1]), \dots, p([m_k])\big).$$

Finally, we define the entropy of a phrase as the entropy of its phrase corpus.

3 Method Analysis

When programmers express themselves in Programmer English, they give both the actual expression (the name) and their subjective interpretation of that expression (the implementation). We therefore analyze each method in two ways: (a) a syntactic analysis concerned with interpreting the name, and (b) a semantic analysis concerned with interpreting the implementation. The input to the analyses is a Java method as found in a Java class file, the output an idealized *method* as defined in Sect. 2.4, consisting of fingerprint, name and semantics.

3.1 Syntactic Analysis of Method Names

Method names are not arbitrarily chosen; they have meaning and structure. In particular, names consist of one or more *words* (which we call fragments) put together to form phrases in Programmer English. We apply natural language processing techniques [6], in particular *part-of-speech tagging*, to reveal the grammatical structure of these phrases.

Decomposition of Method Names. Since whitespace is not allowed in identifiers, the Java convention is to use the transition from lower-case to upper-case letters as delimiter between fragments. For example, a programmer expressing the phrase "get last element" would encode it as `getLastElement`. Since we want to analyze the method names as natural language expressions, we reverse engineer this process to recover the original phrase. This involves *decomposing* the Java method names into fragments.

Since we are focussed on the typical way of expression in Java, we discard names that use any characters except letters and numbers. This includes names using underscore rather than case transition as a delimiter between fragments. Since less than 3% of the methods in the original corpus contain underscores, this has minimal impact on the results. That way, we avoid having to invent ad-hoc rules and heuristics such as *delimiter precedence* for handling method names with underscores *and* case transition. For instance, programmers may mix delimiters when naming test methods (e.g., `test_handlesCornerCase`), use underscores for private methods (e.g., `_findAccount`) or in other special cases (e.g., `processDUP2_X2`). Indeed, nearly half the names containing underscores have an underscore as the first character.

Part-of-speech Tagging. The decomposed method name is fed to our *part-of-speech tagger* (POS tagger), which marks each fragment in the method name with a certain *tag*. Informally, part-of-speech tagging means identifying the correct role of a word in a phrase or sentence.

Our POS tagger for method names is made simple, as the purpose is to provide a proof-of-concept, rather than create the optimal POS tagger for method names in Java. While there exist highly accurate POS taggers for regular English, their performance on Programmer English is unknown. Manual inspection of 500 tagged names taken from a variety of grammatical structures indicates that our POS tagger has an accuracy above 97%.

The POS tagger uses a primitive tag set: **verb**, **noun**, **adjective**, **adverb**, **pronoun**, **preposition**, **conjunction**, **article**, **number**, **type** and **unknown**. Examples of **unknown** fragments are misspellings (”anonimous”), idiosyncracies (”xget”) and composites not handled by our decomposer (”nocando”). Less than 2% of the fragments in the corpus are **unknown**.

A fragment with a **type** tag has been identified as the name of a Java type. We consider a Java type to be in scope for the method name if the type is used in the method signature or body. This implies that the same method name can be interpreted differently depending on context. For instance, the method name `getInputStream` will be interpreted as the two fragments **get-InputStream** if `InputStream` is a Java type in scope of the method name, and as the three fragments **get-input-stream** otherwise. As illustrated by the example, we combine fragments to match composite type names. Type name ambiguity is not a problem for us, since we only need to know that a fragment refers to a type, not which one.

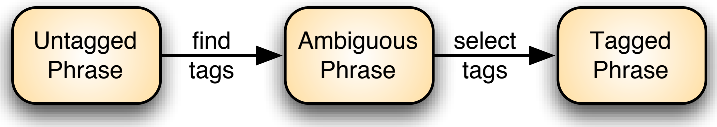


Fig. 2. Overview of the POS tagging process.

The POS tagger operates in two steps, as shown in Fig. 2. First, we determine the range of *possible* tags for the fragments in the phrase, then we *select* a tag for each fragment. WordNet [7] is a core component of the first task. However, since WordNet only handles the four word classes verbs, nouns, adjectives and adverbs, we augment the results with a set of prepositions, pronouns, conjunctions and articles as well. Also, since Programmer English has many technical and specialized terms not found in regular English, we have built a dictionary of such terms. Examples include “encoder” and “optimizer” (nouns) and “blit” and "refactor" (verbs). The dictionary also contains expansions for many common abbreviations, such as "abbrev" for "abbreviation".

The second step of the POS tagging is selection, which is necessary to resolve ambiguity. The tag selector is context-aware, in the sense that it takes into account a fragment's position in a phrase, as well as the possible tags of surrounding fragments. For instance, the fragment **default** is tagged as **adjective** in the phrase **get-default-value**, and as **noun** in the phrase **get-default**. Since we know that method names tend to start with verbs, a fragment is somewhat more likely to be tagged **verb** if it is the first fragment in the phrase. Also, some unlikely interpretation alternatives are omitted because they are not common in programming. For instance, we ignore the possibility of **value** being a verb.

3.2 Semantic Analysis of Method Implementations

The goal of the semantic analysis of the method implemenation is to derive a model of the method's behaviour. This model is an abstraction over the bytecode instructions in the implementation. In Frege's terms, we use the model to capture the programmer's idea of the method.

Attributes. In Sect. 2.4, we defined the semantics of a method m as a list of binary attribute values. The attributes are predicates, formally defined as conditions on Java bytecode. We have hand-crafted the attributes to capture various aspects of the implementation. In particular, we look at *control flow*, *data flow* and *state manipulation*, as well as the *method signature*. In addition, we have created certain attributes that we believe are significant, but that fall outside these categories.

The attributes are listen in Table 1. Each attribute is given a name and a short description. The formal definitions of the attributes range in sophistication, from checking for presence of certain bytecode instructions, to tracing the flow of parameter and field values.

Attribute Dependencies. In our previous work, we used strictly orthogonal attributes [3]. However, this sometimes forces us to choose between coarse and narrow attributes. As an example, we would have to choose between common, but not so distinguishing **Reads field** attribute, and the much more precise and semantically laden **Returns field value**. We therefore allow non-orthogonal attributes in our current work.

Table 2 lists the dependencies between the attributes. We see that they are straight-forward to understand. For instance, it should be obvious that all methods that return a field value must (a) read a field and (b) return a value.

Note that there are more subtle interactions at work between the attributes as well. For instance, **Throws exception** tends to imply **Creates objects**, since the exception object must be created at some point. However, it is not an absolute dependency, as rethrowing an exception does not mandate creating an object.

Table 1. Attributes.

Control Flow
Contains loop There is a control flow path that causes the same basic block to be entered more than once.
Contains branch There is at least one jump or switch instruction in the bytecode.
Multiple return points There is more than one return instruction in the bytecode.
Is recursive The method calls itself recursively.
Same name call The method calls a different method with the same name.
Throws exception The bytecode contains an `ATHROW` instruction.
Data Flow
Writes parameter value to field A parameter value may be written to a field.
Returns field value The value of a field may be used as the return value.
Returns parameter value A parameter value may be used as the return value.
Local assignment Use of local variables.
State Manipulation
Reads field The bytecode contains a `GETFIELD` or `GETSTATIC` instruction.
Writes field The bytecode contains a `PUTFIELD` or `PUTSTATIC` instruction.
Method Signature
Returns void The method has no return value.
No parameters The method has no parameters.
Is static The method is static.
Miscellaneous
Creates objects The bytecode contains a `NEW` instruction.
Run-time type check The bytecode contains a `CHECKCAST` or `INSTANCEOF` instruction.

Table 2. Attribute dependencies.

Contains loop $\Longrightarrow$ **Contains branch**
Writes parameter value to field $\Longrightarrow$ **Writes field** $\wedge \neg$**No parameters**
Returns field value $\Longrightarrow$ $\neg$**Returns void** $\wedge \neg$**Reads field**
Returns parameter value $\Longrightarrow$ $\neg$**Returns void** $\wedge \neg$**No parameters**

Critique. We have constructed the set of attributes under two constraints: our own knowledge of significant behaviour in Java methods and the relative simplicity of our program analysis. While we believe that the attributes are adequate for demonstration, we have no illusions that we have found the optimal set of attributes. A more sophisticated program analysis might allow us to define or approximate interesting attributes such as **Pure function** (signifying that the method has no side-effects). It is also not clear that attributes are the best way to model the semantics of methods — for instance, the structure of implementations is largely ignored. However, the simplicity of attributes is also the strength of the approach, in that we are able to reduce the vast space of possible implementations to a small set of values that seem to capture their essence. This is important, as it facilitates comparing and contrasting the semantics of methods described by different phrases.

3.3 Phrase Semantics

The semantics of a single method captures the programmer's subjective idea of what a method phrase means. When we gather many such ideas, we can approximate the sense of the phrase, that is, its objective meaning. We can group ideas by their concrete method phrases (names) such as **compare-to**, or by more abstract phrases containing tags, such as **find-[type]-by-[noun]**.

Phrase Characterization. Just like other natural language expressions, a phrase in Programmer English is only meaningful in contrast to other phrases. The English word *light* would be hard to grasp without the contrast of *dark*; similarly, we understand a phrase like **get-name** by virtue that it has different semantics from its opposite **set-name**, and also from all other phrases, most of which are semantically unrelated, such as **compare-to**. Since the semantics $[\![p]\!]$ of a phrase p is defined in terms of a list of attribute frequencies (see Sect. 2.4), we can characterize p simply by noting how its individual frequencies deviates from the average frequencies of the same kind.

For a given attribute a, the relative frequency $\xi_a(\mathcal{C}_n)$ for all names $n \in \mathcal{N}$ lies within the boundaries $0 \leq \xi_a(\mathcal{C}_n) \leq 1$. We divide this distribution into five named groups, as shown in Table 3. Each name is 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. This is a convenient way of mapping continuous values to discrete groups, greatly simplifying comparison.

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

3.4 Method Delegation

The use of method delegation in Java programs — invoking other methods instead of defining the behaviour locally — is a challenge for our analysis. The rea-

Table 3. Percentile groups for attribute frequencies.

$< 5\%$	Low extreme
$< 25\%$	Low
25% - 75%	Unlabelled
$> 75\%$	High
$> 95\%$	High extreme

son is that a method implementation that delegates directly to another method exposes no behaviour of its own, and so it "waters down" the semantics of the method name.

There are two simple ways of handling delegation: inlining and exclusion. Inlining essentially means copying the implementation of the called method into the implementation of the calling method. There are several problems with inlining. First, it undermines the abstraction barrier between methods and violates the encapsulation of behaviour. Second, it skews the analysis by causing the same implementation to be analyzed more than once. We therefore prefer exclusion, which means that delegating methods are omitted from the analysis. However, what constitutes delegation is fuzzy: should we ignore methods that calculate parameters passed to other methods, or methods that delegate to a sequence of methods? For simplicity, we only identify and omit single, direct delegation with an equal or reduced list of parameters.

4 Engineering the phrase book

This section describes the engineering efforts undertaken to produce the proof-of-concept phrase book for Java programmers. In particular, we describe how we meet the requirements outlined in Sect. 2.2, and take a closer look at the algorithm used to generate the phrase book.

4.1 Meeting the Requirements

In Sect. 2.2, we mandated that the phrase book be relevant and useful.

Relevance We fulfill the relevance requirement by using a large corpus of Java applications as data for our analysis. This ensures that the results reflect actual real-world practice. The corpus is the same set of applications we used in our previous work [3]. Since many applications rely on third-party libraries, the corpus has been carefully pruned to ensure that each library is analyzed only once. This is important to avoid skewing of the results: otherwise, we cannot be certain that the semantics of a phrase reflects the cross-section of many different implementations.

The corpus consists of 100 open-source applications and libraries from a variety of domains: desktop applications, programmer tools, programming languages, language tools, middleware and frameworks, servers, software development kits, XML tools and various common utilities. Some well-known examples include the Eclipse integrated development environment, the Java software development toolkit, the Spring application framework, the JUnit testing framework, and the Azureus bittorrent client[3]. Combined, the applications in the corpus contain more than one million methods.

Usefulness In order to produce a phrase book that is as useful as possible, we want the phrase book to be short, easy to read, and containing only the most useful phrases. Here, we present our translation of the qualitative usefulness requirements into quantitative ones.

– *Validity.* Each phrase must represent at least 100 methods.
– *Precision.* Intuitively, precision means how consistently a phrase refers to the same semantics. Since entropy measures the independence of attributes, entropy is an inverse measurement of precision. Each phrase representing a refinement of another phrase must therefore lead to *decreased* entropy, corresponding to *increased* precision.
– *Ubiquity.* Each phrase must be present in at least half of the applications in the corpus.

Tweaking the actual numbers in these criteria allows us to control the size of the phrase book. The values we have chosen yields a phrase book containing 364 phrases. The ideal size of the phrase book is a matter of taste; we opt for a relatively small one compared to natural-language dictionaries.

4.2 Generation Algorithm

Below, we present and explain the pseudo-code (Fig. 3) for the algorithm that automatically generates the phrase book. Note that the pseudo-code glosses over many details to highlight the essentials of the algorithm. For brevity, we omit definitions for functions that are "self-explanatory". The syntax is influenced by Python, meaning that indentation is significant and used to group blocks.

The pseudo-code outlines a fairly simple recursive algorithm. The driving function is refine, which generates a refinement of a phrase. Note that each phrase implicitly defines a corpus of methods, so that a refinement of a phrase also means a narrowing of the corpus.

First, we iterate over the tags in our tag set (see Sect. 3.1). For each tag, we create a new phrase representing a refinement to only the methods whose names satisfy the new tag. We demand that this refinement be useful, or we ignore the

[3] Azureus is currently the most downloaded and actively developed application from SourceForge.net.

```
refine(phrase):
  for tag in tags:
    t-phrase = phrase-append(phrase, tag)
    if useful(t-phrase, phrase):
      used-phrases = ()
      for f-phrase in fragment-phrases(p, tag):
        if useful(f-phrase, t-phrase):
          used-phrases.add(f-phrase)
          write-entry(f-phrase)
          refine(f-phrase)
      r-phrase = mark-special(t-phrase)
      if useful(r-phrase, phrase):
        write-entry(r-phrase)
        refine(r-phrase)
```

Fig. 3. Pseudo-code for the phrase book generation algorithm

entire tag. The refinement is useful if it meets the criteria of the useful function. This function embodies the criteria discussed in Sects. 2.2 and 4.1.

If the refinement is useful, we try to find even more useful refinements using fragments instead of the tag. Assume that we are calling expand on the phrase **get-***. We expand the phrase with the tag **noun**, yielding the new phrase **get-[noun]-***. Finding the new phrase to be useful, we generate more concrete refinements such as **get-name-*** and **get-customer-***. If they are useful, we call the write-entry function, which generates a description that is included in the phrase book, and recurse, by calling refine on the concrete refinement. Finally, we examine the properties of the corpus of remnant methods; those that match the tag phrase, but are not included in any of the useful concrete refinements. We say that these methods are captured by a special phrase r-phrase. The r-phrase is equal to the t-phrase, except that it potentially captures fewer method names, and hence might represent a smaller corpus. For instance, if **get-name-*** is useful and **get-customer-*** is not, then r-phrase captures the phrases captured by **get-[noun]-***, except those also captured by **get-name-***. If r-phrase is useful, it is included in the phrase book. Note that if no useful concrete refinements are found, r-phrase degenerates to t-phrase.

5 Results

While the phrase book has been designed for brevity, it is still much too large to be included in this paper. We therefore present some excerpts highlighting different aspects. The full version is available at `http://phrasebook.nr.no`. We also take a look at the distribution of grammatical structures, and using the phrase book to guide naming.

Terminology. Table 4 explains the basic terminology used in the phrase book. In addition, we use the modifier *comparatively* to indicate that the frequency is low despite being in the higher quantiles, or high despite being in the lower quantiles. For instance, a phrase might denote methods that call themselves recursively *more often* than average methods, even if the actual frequency might be as low as 0.1.

Table 4. Phrase book terminology.

Phrase	*Meaning*
Always	The attribute value is always 1.
Very often	Frequency in the high extreme percentile group.
Often	Frequency in the high percentile group.
Rarely	Frequency in the low percentile group.
Very rarely	Frequency in the low extreme percentile group.
Never	The attribute value is always 0.

Example Entry. To illustrate how the data uncovered by our analysis is presented in the phrase book, we show the entry for the phrase **find-[type]**. It is an interesting example of a slightly abstract phrase with a clear meaning.

find-[type]. These methods very often contain loops, use local variables, have branches and have multiple return points, and often throw exceptions, do runtime type-checking or casting and are static. They rarely return void, write parameter values to fields or call themselves recursively.

Each entry in the phrase book describes what signifies the corpus of methods captured by the phrase; that is, how it differs from the average (Sect. 3.3). We find no surprises in the distinguishing features of **find-[type]**; in fact, it is a fairly accurate description of a typical implementation such as:

```
Person findPerson(String ssn) {
  Iterator itor = list.iterator();
  while (itor.hasNext()) {
    Person p = (Person) itor.next();
    if (p.getSSN().equals(ssn)) {
      return p;
    }
  }
  return null;
}
```

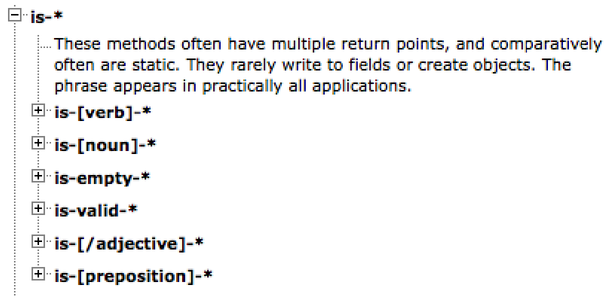
We iterate over a collection of objects (**Contains loop**), cast each object to its proper type (**Run-time type check**) and store it in a variable (**Local**

assignment), and terminate early if we find it (**Multiple returns**). A common variation would be to throw an exception (**Throws exception**) instead of returning `null` if the object could not be found.

Refinement. As explained in Sect. 4.1, the phrase book is engineered to yield useful entries, understood as valid, precise and ubiquitous ones. The generation algorithm has been designed to prefer concrete phrases over abstract ones, as long as the criteria for usefulness are fulfilled.

One effect of this strategy is that the everyday “cliché” methods `equals`, `hashCode` and `toString` defined on `Object` are not abstracted: they emerge as the concrete phrases **equals**, **hash-code** and **to-String**. This is not surprising, as the names are fixed and the semantics are well understood.

The algorithm's ability to strike the right balance between concrete and abstract phrases is further illustrated by the branch for the **is-*** phrase, shown in Fig. 4.



is-*

These methods often have multiple return points, and comparatively often are static. They rarely write to fields or create objects. The phrase appears in practically all applications.

is-[verb]-*

is-[noun]-*

is-empty-*

is-valid-*

is-[/adjective]-*

is-[preposition]-*

Fig. 4. The **is-*** branch of phrases.

We see that the algorithm primarily generates abstract refinements for **is-***; one for each of the tags **verb**, **noun**, **adjective** and **preposition**. However, in the case of **adjective**, two concrete instances are highlighted: **is-empty-*** and **is-valid-***. This matches nicely with our intuition that these represent common method names. We write [**/adjective**] for the subsequent phrase to indicate that it captures adjectives except the preceding “empty” and “valid”.

Grammar. We find that the vast majority of method phrases have quite degenerate grammatical structures. By far the most common structure is [**verb**]-[**noun**]. Furthermore, compound nouns in Programmer English, as in regular English, are created by juxtaposing nouns. The situation becomes even more extreme when we collapse these nouns into one, and introduce the tag **noun+** to represent a compound noun. The ten most common grammatical structures are listed in Table 5.

Table 5. Distribution of grammatical structures.

Structure	*Instances*	*Percent*
[**verb**]-[**noun+**]	422546	39.45%
[**verb**]	162050	15.13%
[**verb**]-[**type**]	78632	7.34%
[**verb**]-[**adjective**]-[**noun+**]	74277	6.93%
[**verb**]-[**adjective**]	28397	2.65%
[**noun+**]	26592	2.48%
[**verb**]-[**noun+**]-[**type**]	18118	1.69%
[**adjective**]-[**noun+**]	15907	1.48%
[**noun+**]-[**verb**]	14435	1.34%
[**preposition**]-[**type**]	13639	1.27%

Guidance. Perhaps the greatest promise of the phrase book is that it can be used as guidance when creating and naming new methods. Each description could be translated to a set of rules for a given phrase. An interactive tool, e.g., an Eclipse plug-in, could use these rules to give warnings when a developer breaches them.

As an example, consider the phrase **equals**, which the phrase book describes as follows:

> **equals.** These methods very often have parameters, call other methods with the same name, do runtime type-checking or casting, have branches and have multiple return points, and often use local variables. They never return field values or return parameter values, and very rarely return void, write to fields, write parameter values to fields or call themselves recursively, and rarely create objects or throw exceptions. The phrase appears in most applications.

The extreme clauses are most interesting, because they represent the clearest characteristics for the phrase. For instance, no programmer contributing to the corpus has ever let an **equals** method return a value stored in a field — a strong suggestion that you might not want to do so either! However, we note that the phrase book reflects the actual use of phrases, not the ideal use. This means that the description might capture systematic implementation problems; i.e., common malpractice for a given phrase. We might look for clues in the negative clauses, indicating rare — even suspicious — behaviour. For instance, we see that there are **equals** methods that create objects and throw exceptions, which might be considered dubious. More severely, recursion in an **equals** method sounds like a possible bug. Indeed, inspection reveals an *infinite recursive loop* bug in an **equals** method in version 1.0 of Groovy[4].

We conclude that the rules uncovered by the phrase book appear to be useful as input to a naming-assistance tool. However, the rules might need to be tightened somewhat, to compensate for fallible implementations in the corpus.

[4] `http://groovy.codehaus.org/`

After all, the corpus reflects the current state of affairs for naming, and the aim of a naming-assistance tool would be to improve it.

6 Related Work

We build on our previous work [3], which defined semantics for *action verbs*, the initial fragment of method names. We summarized the findings in *The Programmer's Lexicon*, an automatically generated description of the most common verbs in a large corpus of Java applications. The distinguishing characteristic of our work, is that we compare the names and semantics of methods in a large corpus of Java applications.

Other researchers have analyzed Java applications in order to describe typical Java programmer practice. Collberg et al. [8] present a large set of low-level usage statistics for a huge corpus of Java programs. Examples of statistics included are the number of subclasses per class, the most common method signatures and bytecode frequencies. Baxter et al. [9] have similar goals, in using statistics to describe the anatomy of real-world Java programs. In particular, they investigate the claim that many important relationships between software artifacts follow a "power-law" distribution. However, none of these statistics are linked to names.

There have also been various kinds of investigations into identifiers, traditionally in the context of program comprehension. Lawrie et al. [10] study how the quality of identifiers affect the understanding of source code. Caprile and Tonella [11] investigate the structure of function identifiers as found in C programs. They build a dictionary of identifier fragments and propose a grammar for identifiers, but make no attempt at defining identifier semantics. Antoniol et al. [12] find that the names used in programming evolve more slowly than the structures they represent. They argue that the discrepancy is due to lack of tool support for name refactoring. In this work, names are linked to structures, but not the semantics of the structures.

Lately, more interest can be seen in investigating and exploiting name semantics. Singer et al. [13] share our ambition in ascribing semantics to names based on how they are used. They analyze a corpus of real-world Java applications, and find evidence of correlation between type name suffixes (nouns) and some of the micro patterns described by Gil and Maman [14]. Micro patterns are formal, traceable conditions defined on Java types.

Pollock et al. [15] investigate various ways of utilizing "natural language clues" to guide program analysis. Shepherd et al. [16] apply method name analysis to aid in aspect mining. In particular, they investigate the relationships between verbs (actions) and nouns (types) in programs. The scattering of the same verb throughout a program is taken as a hint of a possible cross-cutting concern. Finally, Ma et al. [17] use identifier fragments to index software repositories, to assist in querying for reusable components. These works involve exploiting implicit name semantics, in that relationships between names are taken to be meaningful. However, what the semantics are remains unknown. Our work is different, in that we want to explicitly model and describe the semantics of each name.

7 Conclusion

The names and implementations of methods are mutually dependent on each other. The phrase book contains descriptions that captures the objective *sense* of the phrases, that is, the common understanding among Java programmers of what the phrases mean. We arrived at the sense by correlating over a million method names and implementations in a large corpus of Java applications. By using attributes, defined as predicates on Java bytecode, we modeled the semantics of individual methods. By aggregating methods by the phrases that describe them, we derived the semantics of the phrases themselves. From the semantics, we generated the textual descriptions gathered in the phrase book.

We believe that further investigation into the relationship between names and implementations can yield more valuable insight and contribute to improved naming. Currently, we are refining our model of the semantics of methods, in order to make it more sophisticated and precise. This will allow us to more accurately describe the meaning of the phrases. An obvious enhancement is to use a state-of-the-art static analysis tool to provide a richer, more descriptive set of attributes. We are also considering developing a model for the semantics that better captures the structure of the implementations. At an abstract level, it might be possible to identify machine-traceable *patterns* for method implementations. Inspired by Singer et al. [13], then, we might look for correlation between names and these patterns.

While we have shown that names do have grammatical structure, we believe that the potential for natural language expression in names is under-utilized. Indeed, by far the most common structure is the simple **[verb]-[noun]** structure. Longer, more complex names gives the possibility of much more precise descriptions of the behaviour of methods. Improved tool support for verifying name quality might motivate programmers to exploit this possibility to a greater extent than at the present.

We are working on transforming the results presented in this paper into a practical tool, supporting a much richer set of naming conventions than adherence to simple syntactic rules such as the camel case convention. The tool will warn against dissonance between name and implementation, and suggest two paths to resolution: 1) select a more appropriate name from a list proposed by the tool, or 2) perform one or more proposed changes to the implementation.

In a somewhat longer timeframe, the tool could be extended to support grammatical conventions as well. An example would be to warn against mixing verbose and succinct naming styles. One might debate whether it is better to explicitly mention types in method names (e.g., `Customer findCustomerByOrder(Order)`) or not (e.g., `Customer find(Order)`), but to mix both styles in the same application is definitely confusing. Tool support could help achieve grammatical consistency within the application.

References

1. Edwards, J.: Subtext: uncovering the simplicity of programming. [18] 505–518
2. Frege, G.: On sense and reference. In Geach, P., Black, M., eds.: Translations from the Philosophical Writings of Gottlob Frege. Blackwell (1952) 56–78
3. Høst, E.W., Østvold, B.M.: The programmer's lexicon, volume I: The verbs. In: SCAM '07: Proceedings of the Seventh IEEE International Working Conference on Source Code Analysis and Manipulation, Washington, DC, USA, IEEE Computer Society (2007) 193–202
4. Wittgenstein, L.: Philosophical Investigations. Prentice Hall (1973)
5. Cover, T.M., Thomas, J.A.: Elements of Information Theory. 2nd edn. Wiley Series in Telecommunications. Wiley (2006)
6. Manning, C.D., Schuetze, H.: Foundations of Statistical Natural Language Processing. MIT Press (1999)
7. Fellbaum, C.: WordNet: An Electronic Lexical Database. MIT Press (1998)
8. Collberg, C., Myles, G., Stepp, M.: An empirical study of Java bytecode programs. Software Practice and Experience **37**(6) (2007) 581–641
9. Baxter, G., Frean, M., Noble, J., Rickerby, M., Smith, H., Visser, M., Melton, H., Tempero, E.: Understanding the shape of Java software. In: Proceedings of the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006, October 22-26, 2006, Portland, Oregon, USA, ACM (2006) 397–412
10. Lawrie, D., Morrell, C., Feild, H., Binkley, D.: What's in a name? a study of identifiers. In: Proceedings of the 14th International Conference on Program Comprehension (ICPC 2006), 14-16 June 2006, Athens, Greece, IEEE Computer Society (2006) 3–12
11. Caprile, B., Tonella, P.: 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, IEEE Computer Society (1999) 112–122
12. Antonial, G., Guéhéneuc, Y.G., Merlo, E., Tonella, P.: Mining the lexicon used by programmers during sofware [sic] evolution. Proc. of the International Conference on Software Maintenance (ICSM) (2007) 14–23
13. Singer, J., Kirkham, C.: Exploiting the correspondence between micro patterns and class names. In: SCAM '08: Proceedings of the Eight IEEE International Working Conference on Source Code Analysis and Manipulation, IEEE Computer Society (2008) To appear.
14. Gil, J., Maman, I.: Micro patterns in Java code. [18] 97–116
15. 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.: PASTE, ACM (2007) 15–16
16. Shepherd, D., Pollock, L.L., Vijay-Shanker, K.: Towards supporting on-demand virtual remodularization using program graphs. In Filman, R.E., ed.: AOSD, ACM (2006) 3–14
17. Ma, H., Amor, R., Tempero, E.D.: Indexing the Java API using source code. In: Australian Software Engineering Conference, IEEE Computer Society (2008) 451–460
18. OOPSLA 2005. In: Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2005, October 16-20, 2005, San Diego, CA, USA, ACM (2005)