# A Survey of the Forms of Java Reference Names

Simon Butler, Michel Wermelinger and Yijun Yu
Computing and Communications Department, The Open University
Milton Keynes MK7 6AA, United Kingdom

***Abstract*—The readability of identifiers is a major factor of program comprehension and an aim of naming convention guidelines. Due to their semantic content, identifiers are also used in feature and bug location, among other software maintenance tasks. Looking at how names are used in practice may lead to insights on potential problems for comprehension and for programming support tools that process identifiers.**

**Class and method names are already well represented in the literature. This paper presents an investigation of Java field, formal argument and local variable names, which we collectively call reference names. These names cannot be ignored because they constitute over half the unique names and almost 70% of the name declarations in the corpus investigated.**

**We analysed the forms of 3.5 million reference name declarations in 60 well known Java projects, examining the phrasal structure of names composed of known words and acronyms. The structures found in practice were evaluated against those given in the literature. The use of unknown abbreviations and words, which may pose a problem for program comprehension, was also identified. Based on our observations of the rich diversity of reference names, we suggest issues to be taken into account for future academic research and for improving tools that rely on names as sources of information.**

## I. Introduction

*The name you can name isn't the real name.* – Laozi

Field, formal argument and local variable names, henceforth collectively called *reference names*, constitute 52% of the unique names, and 69% of all declarations found in a corpus of 60 FLOSS Java projects [1], and are a potentially rich source of information for the tools that support program comprehension, including code search and feature location.

As Laozi observes, in general it is hard to choose a name to reflect the real meaning. Identifier naming conventions [2][3] provide developers with guidelines for composing names. The guidelines can be complex, particularly for reference names, providing developers with plenty of choice about the form of name they create, including: phrase-like names containing words, abbreviations and acronyms; isolated abbreviations; acronyms derived from type names; and specialised, single-letter abbreviations for generic or short-lived identifiers with well-understood roles. Developers are free to ignore naming conventions and to create identifier names as they please. Consequently the readers of source code, including program comprehension tools that rely on the content of names, have only an outline of the forms names might take, and the possibility of being surprised.

Liblit *et al.* identified, through observation, patterns of naming they referred to as *metaphors* where they considered names to be phrasal utterances [4]. Their metaphors are often used as a starting point by those analysing names for program comprehension [5][6][7]. However, the extent to which Liblit *et al.*'s metaphors are used by developers has not been established for all *species* of identifier — by species we mean class, method, field, etc. Work on Java method names [8] and class names [9] show that the metaphors are used extensively but developers also create names with unanticipated phrasal structures, both simple and complex. One may expect developers to be similarly inventive with reference names.

In this paper we seek to establish the forms of reference names created by developers and the extent to which the various forms are used. As mentioned earlier, naming conventions suggest a choice of name *components*, e.g. dictionary words, acronyms, and abbreviations. Our first research question is:

RQ1 What components do developers use to create reference names, and to what extent?

As part of the above question, we want to know to which degree recognizable abbreviations are used, which might be readily comprehended by humans but not by tools without further processing, namely abbreviation expansion.

The literature argues that the use of natural language phrases and metaphors can ease comprehension. We wish to know:

RQ2 What phrasal structures do developers use in reference names, and how are they related to Liblit *et al.*'s metaphors?

The remainder of the paper is structured as follows. Section II describes our method, Sections III and IV respectively report and discuss our survey results, Section V relates previous work to ours, and Section VI draws conclusions.

## II. Methodology

In this study we investigate only Java source code written in English because it is the most widely used natural language in software development.

The strong typing of Java offers two features that simplify the identification of the role of a reference name. Firstly, in Java all boolean identifiers are declared using the `boolean` primitive or the `Boolean` object type, unlike C for example, where numeric values may be used as booleans. This allows us to accurately distinguish boolean identifiers to investigate Liblit *et al.*'s observation that they differ in structure to non-boolean names [4]. Secondly, analysing the types of reference names in Java is feasible because, with the exception of the reflection API, there are no function pointers in Java, and there is a clear distinction between actions and entities. All

source code in this study pre-dates the introduction of method references in Java 8.

### A. The Dataset

For this survey we used INVocD [1], a freely available database of all names occurring in 60 well-known Java FLOSS projects from 2006–2010[1]. Among other information, INVocD records for each declared name its species and its type.

The corpus for this study is the bag, i.e. a set with duplicate elements, of all reference name declarations in INVocD: 626,262 fields, 1,556,931 formal arguments, and 1,319,071 local variables. We survey declarations instead of just names because we wish to distinguish names declared with different species or types.

Each declaration indicates the need for a name, and the developer has a free choice. If the developer reuses a particular name, it indicates preference for certain identifier forms, phrasal structures or metaphors. The reuse of names is indeed substantial. The declarations given above introduce only 272,228 unique field names, 81,201 unique formal argument names and 169,428 unique local variable names.

To capture such preferences, the corpus is a bag instead of a set, i.e. *all* declarations are considered, even of the same name with the same type and species. In this way, for a program that declares 100 integer local variables, one named `xpto` and the rest `i`, we obtain 99% of expected name forms (namely `int i`), whereas considering only unique names or unique declarations would lead to a distorted figure of 50%, when in fact the developer made 100 choices, only one of which deviated from established guidelines.

For names to be considered phrases they should, in general, consist of sufficient 'words' to form a phrase and not so many as to form multiple phrases. Lawrie *et al.* [10] define a *hard word* as one divided from its neighbour by a typographical boundary, such as a change of case or a separator character, and a *soft word* as an abbreviation or dictionary word that may or may not be a hard word. For example, the name `ANEWARRAY` (Rhino), representing a Java bytecode operation, is composed of one hard word and three soft words: 'a', 'new', and 'array'. To avoid any confusion, we henceforth use the term *token* to mean soft word and the term *word* to mean a dictionary word.

Lawrie *et al.* found that Java names have 3.4 tokens on average. These are mean values for all the unique names found in the code they investigated: there was no breakdown by name species and no measure of central tendency. INVocD already provides each name's tokens, as computed by INTT [11], a freely available identifier tokeniser. Table I shows the distribution of the length of the unique names in INVocD as number of tokens. Field name length is similar to Lawrie *et al.*'s mean, while formal argument and local variable names tend to be much shorter. The longest name, with 39 tokens, is one of a number of long names given to string constants in Eclipse used as keys in resource bundles. Those names form multiple phrases, which we discuss in Section IV.

[1]See http://www.facetus.org.uk/corpus.html for the list of projects.

TABLE I
DISTRIBUTION OF LENGTH (IN TOKENS) OF UNIQUE REFERENCE NAMES

| | Field | Formal Argument | Local Variable |
|---|---|---|---|
| **Minimum** | 1 | 1 | 1 |
| **1st Quartile** | 2 | 2 | 2 |
| **Median** | 3 | 2 | 2 |
| **Mean** | 3.1 | 2.3 | 2.4 |
| **3rd Quartile** | 4 | 3 | 2 |
| **Maximum** | 39 | 10 | 12 |

As mentioned earlier, we need to examine boolean names separately. The proportion of unique reference names declared as `boolean` or `Boolean` in the subject projects is shown in Table II. There is considerable variation between the projects. We found that 1.1% to 15.5% of unique field names are declared as booleans in the projects investigated, and over 20% of unique formal arguments in Ant, OpenProj and Vuze.

TABLE II
DISTRIBUTION OF PROPORTIONS OF UNIQUE BOOLEAN REFERENCE NAMES

| | Field | Formal Argument | Local Variable |
|---|---|---|---|
| **Minimum** | .011 | .024 | .017 |
| **1st Quartile** | .067 | .079 | .046 |
| **Median** | .083 | .113 | .063 |
| **Mean** | .086 | .112 | .063 |
| **3rd Quartile** | .102 | .136 | .079 |
| **Maximum** | .155 | .218 | .118 |

Java does not permit the use of punctuation in names, such as the use of apostrophes to identify possessive forms and contractions of negated modal verbs, e.g. `ER_CANT_CREATE_URL` (Xalan). Whilst negated modal verbs are not extensively used, they are easily recognised and expanded, allowing them to be tagged correctly (Section II-C) to reduce noise. We therefore expanded, prior to any further processing, all non-apostrophised negated modal verbs to their two-word form, e.g. 'wont' to 'will not'. Although 'cant' (meaning hypocritical and sanctimonious talk, among other things) and 'wont' (meaning accustomed) are English words, we still interpret them as negated modal verbs, as it is the most likely use in source code identifiers. We did not attempt to identify or expand possessive forms of nouns.

### B. Partitioning Names

Our principal interest is the phrasal structure of reference names, but not all reference names are composed of words. Naming conventions, such as those in JLS ('Java Language Specification' [2]) and EJS ('The Elements of Java Style' [3]), direct developers to use a mixture of well understood single letter names, acronyms derived from the type name, other acronyms[2], abbreviations, words, and multi-token names that combine the previous three categories. Tokens containing digits may be known acronyms, such as 'MP3', otherwise they are categorised as unrecognised.

[2]We use a definition of *acronym* that includes initialisms such as HTML.

TABLE III
CIPHERS AND THEIR CORRESPONDING TYPES

| Cipher(s) | Type(s) | Source |
|---|---|---|
| b | byte, Byte | JLS |
| c | char, Character | JLS, EJS |
| d,e | char, Character | EJS |
| d | double, Double | JLS |
| e | Exception | JLS |
| f | float, Float | JLS |
| g | Graphics | EJS |
| i,j,k | int, Integer | JLS |
| l | long, Long | JLS, EJS |
| o | Object | JLS, EJS |
| s | String | JLS, EJS |
| v | a value of some type | JLS |
| x,y,z | any numeric type | EJS |

The variety of name forms means that they cannot all be analysed together. Accordingly we partition reference name declarations ($N$) into the following bags for each species:

- $C$ contains *ciphers*, i.e. well-known or conventional single letter abbreviations (Table III);
- $T$ contains acronyms derived from type names;
- $P$ contains names consisting only of 'processed' components: dictionary words, known technical terms and acronyms, and an optional redundant prefix discussed in Section II-C;
- $U$ contains names with at least one 'unprocessed' component, i.e. an abbreviation or an unrecognised token.

Names are partitioned in the following order. A name is first tested to determine whether it is a cipher from Table III, and whether it is of the correct type. Table III shows that we have widened the JLS and EJS definitions of permitted types to include the Java v5 classes that wrap primitive types such as `Integer`, so that the declarations `for(int i; ...)` and `for(Integer i; ...)` are both considered ciphers.

Should the name fail that test, we check if it is a type acronym as suggested in the JLS, i.e. an initialism derived from the declared type name, e.g. `FileWriter fw`. With these tests, `Iterator i` is in $T$, not in $C$.

Names not meeting the requirements for $C$ and $T$ are partitioned into $P$ and $U$: those that consist of recognised prefixes, words and technical terms are assigned to $P$, the rest to $U$. To support the creation of $P$ and $U$ we 1) took the SCOWL [12] word lists (up to size 80) used to create the dictionaries for GNU Aspell[3], 2) added lists of computing terms, Java acronyms, and terms taken from the AMAP project[4] [13] and our own work[5], and 3) divided the lists into separate dictionaries of words, abbreviations, and acronyms.

Table IV shows the distribution of declarations in each partition in the projects analysed. Most declarations are in $P$, e.g. at least 47.8% of each project's field name declarations use only English words and known prefixes and technical terms.

[3]http://aspell.net/

[4]http://msuweb.montclair.edu/~hillem/AMAP.tar.gz

[5]The word lists we used are available as part of the mdsc spell checking library https://github.com/sjbutler/mdsc/

TABLE IV
DISTRIBUTION OF PROPORTIONS OF DECLARATIONS IN EACH PARTITION

| | | Field | Argument | Variable |
|---|---|---|---|---|
| $C$ | **Minimum** | .000 | .001 | .012 |
| | **1st Quartile** | .000 | .043 | .056 |
| | **Median** | .001 | .066 | .075 |
| | **Mean** | .002 | .065 | .087 |
| | **3rd Quartile** | .002 | .086 | .110 |
| | **Maximum** | .015 | .193 | .240 |
| $T$ | **Minimum** | .000 | .002 | .006 |
| | **1st Quartile** | .001 | .016 | .036 |
| | **Median** | .004 | .033 | .054 |
| | **Mean** | .007 | .037 | .060 |
| | **3rd Quartile** | .009 | .050 | .079 |
| | **Maximum** | .043 | .129 | .209 |
| $P$ | **Minimum** | .478 | .267 | .418 |
| | **1st Quartile** | .748 | .715 | .642 |
| | **Median** | .812 | .781 | .693 |
| | **Mean** | .807 | .767 | .692 |
| | **3rd Quartile** | .862 | .083 | .766 |
| | **Maximum** | .961 | .965 | .876 |
| $U$ | **Minimum** | .039 | .027 | .045 |
| | **1st Quartile** | .129 | .081 | .121 |
| | **Median** | .172 | .112 | .155 |
| | **Mean** | .185 | .130 | .161 |
| | **3rd Quartile** | .238 | .150 | .183 |
| | **Maximum** | .522 | .711 | .539 |

A consequence of the method used to create partitions is that all names containing spelling mistakes and readily understood neologisms not in our lists will also be assigned to $U$. Therefore, $U$ has names that may contain English words, but require further processing, such as abbreviation expansion, spell checking and neologism checking. Such names would be a source of noise in our phrasal analysis.

We do not expand abbreviations in this paper. Abbreviations may have more than one plausible expansion and determining the correct one requires assumptions of the phrasal structure of names that pre-empts any investigation of phrasal structure.

### C. PoS Tagging

In previous work [9], we analysed the composition of Java class names in terms of the parts of speech (PoS) of their component words. In that investigation we trained a model for the Stanford Log-linear PoS tagger [14] on a corpus of Java class names. At the start of the current study we used that class names model with v3.4.1 of the Stanford tagger to PoS tag a set of Java field names. Manual inspection of the tagged field names showed an error rate around 28%, some 15% greater than observed when tagging class names. We decided to train a new model on field names to see if that performed better.

We extracted 30,000 unique field names at random from the database. A training set of 29,894 field names manually tagged by the first author, a native speaker of English, was created — 106 names were discarded because they could not be PoS tagged. Typically the discarded names consisted of one or two abbreviations that were either ambiguous or unrecognised, or incomprehensible combinations of words and abbreviations or neologisms. Examples include `TRGDFTRT` (Derby),

icSigPs2CRD2Tag (JDK), and WEAVEMESSAGE_ANNOTATES (AspectJ). The training set was used to obtain a model for the Stanford PoS tagger. A further 5,000 field names were manually tagged to provide a test set. In addition a smaller test set of 1,000 boolean field names was created to measure the performance of the tagger model on boolean names.

While tagging the training and test sets, we removed redundant prefixes from names because they have no parallel in natural language and cannot be reasonably tagged by the Stanford PoS tagger. For example, some field names are prefixed with `f` for 'field' or `m` for 'member'. Single letters are also occasionally used to represent primitive types in a manner similar to Hungarian Notation [15], including the letters `b`, `c`, `d`, `f`, `i`, `l` and `o`, standing for *boolean/byte*, *char*, *double*, *float*, *int*, *long* and *object* respectively. The proportion of field names with redundant prefixes varies according to the naming style adopted by project developers. In many projects these prefixes are not used, but in a few projects the use of prefixes is conventional, particularly to indicate the role of the name, e.g. the use of `f` to prefix mutable fields in JUnit.

Using the Stanford PoS tagger in its test mode, we found the trained model tagged 85.4% of the field names in the test set correctly (94.5% of individual tokens) and 83.0% of the test boolean field names (93.2% of individual tokens).

Formal argument names appear similar in structure to field names. Indeed, in some naming styles, formal arguments for constructors and mutator methods have identical names to fields [3]. Rather than undertake the potentially unnecessary work of creating a PoS tagger model for formal arguments, the first author hand tagged a test set of 5,000 formal argument names extracted at random from the database. The field name PoS tagger tags 91.7% of formal argument names in the test set correctly and 96.0% of individual tokens.

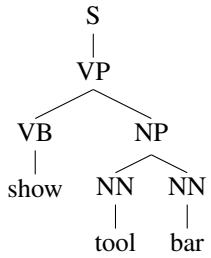
As with field names, we also removed the redundant prefixes from formal arguments used in the test set. The prefixes `p` and `m` are used in combination to distinguish between parameters and members with the same name, e.g. the constructor of `StandardPropertyHandler` (Freemind) has the formal argument `pPropertyName` used to set the field `mPropertyName`.

Local variable names are also similar to field and formal argument names. The process was repeated to create a test set of 4,984 local variable names. According to the Stanford PoS tagger test mode, 90.3% of local variable names and 95.4% of individual tokens in the test set were tagged correctly.

After the tests, the PoS tagger model trained with field names was used to tag all names in the $P$ bags of field, formal argument and local variable name declarations. Where a redundant prefix was found, it was removed and the remainder of the tokenised name tagged by the PoS tagger. The prefix was then added back to the beginning of the tagged string with the tag *RD* for ReDundant.

### D. Phrasal Analysis

We use the Stanford Parser v3.4.1 [16] to identify the phrasal structure of names in the $P$ partition to answer RQ2. The Stanford Parser analyses a PoS tagged string and outputs a phrase structure tree. For example, the name `showToolBar` (BlueJ) is PoS tagged as *show/VB tool/NN bar/NN* (where *NN* is the Penn Treebank PoS tag for noun[6], and *VB* the tag for a verb), which the Stanford parser renders into the phrase tree (S (VP (VB show) (NP (NN tool) (NN bar)))) or:




where S represents a declarative statement, NP a noun phrase and VP a verb phrase. Few of the trees returned by the parser contain clausal elements such as S and SINV (representing a subject-auxiliary inversion like "Is the list empty?"), so we ignore these and treat the top-level phrasal elements as summarising the phrase structure of the name. Our example, `showToolBar`, is summarised as a verb phrase.

A name with a more complex structure is summarised using the two top-level phrases. For example, `entryKeyNotInMyMap` (Polyglot) has the phrase structure tree (FRAG (NP (NN entry) (NN key)) (PP (RB not) (IN in) (NP (PRP$ my) (NN map)))) and is summarised as NP PP, where IN is a preposition, PRP$ a possessive personal pronoun, PP a prepositional phrase, and RB an adverb.

### E. Use of Known Abbreviations

Abbreviations may be truncations (e.g. `impl` for 'implementation'), the elision of letters (e.g. `ctxt` for 'context') or multi-word abbreviations (e.g. `regex` for 'regular expression') [13].

In RQ1 we seek to identify tokens that can be recognized as abbreviations, even though their accurate expansion might not be trivial. We define a known abbreviation as a token found in our abbreviation dictionary, formed from the lists of abbreviations a) extracted from SCOWL, b) with known expansions from the AMAP project, and c) compiled from our own observations of names[5].

### F. Threats to Validity

In addition to the PoS tagger model accuracy described above, there are threats to construct and external validity.

Phrase structure grammars are context free, so, while their use allows the recognition of the aggregation of types of words into grammatically coherent groups, there is no guarantee that the groups are meaningful. Whilst 'The cat sat on the mat' is semantically correct, exchanging the nouns creates an absurdity that is also grammatically correct. Accordingly, there is a threat to *construct validity* from an underlying assumption that the developers have created meaningful rather than absurd names. However, the experimental technique cannot distinguish between the two.

[6]Throughout this paper we use the Penn Treebank notation for PoS tags and phrases (ftp://ftp.cis.upenn.edu/pub/treebank/doc/tagguide.ps.gz).

A minor threat to construct validity arises from our choices of acronym, cipher and word lists used to partition declarations into the $C$ and $P$ categories. Those lists may not coincide with the vocabulary used by the developers of all the projects surveyed — particularly the domain-specific terms and acronyms used. Consequently, some names may have been assigned to the $U$ partition, resulting in a reduction of the size of $P$. A further concern is that EJS specifies the use of the ciphers `x`, `y` and `z` for coordinates. As there is no direct type correspondence, for this survey we widened the definition to accept any numeric type.

Another threat results from the order of tests. Abbreviations such as `ID` that are also English words ('id' is a psychoanalytical term) are recognised as words rather than abbreviations, and the name will be put in a potentially different partition from the developer's intended meaning of the token.

Threats to *external validity* arise because we constrained our experiment in two dimensions. First, we analysed only projects developed in Java, prior to v8, to take advantage of its strong typing; and, secondly, we analysed projects where names were constructed using English words. Accordingly we cannot be sure that our findings may be applied to less strongly typed programming languages, or that developers who create names using languages other than English use a similar phrasal structure. Moreover, several of the issues discussed in Section IV do not apply to languages other than English.

## III. Results

### A. Name Components (RQ1)

The components of a name are its tokens. Each project's tokens were partitioned into 6 sets (not bags), for each name species. For example, the local variable names in Tomcat are composed of 12 ciphers ($\mathbb{C}$), 121 type acronyms ($\mathbb{T}$), 8 redundant prefixes ($\mathbb{R}$), 1206 English words and known acronyms ($\mathbb{W}$), 104 recognised abbreviations ($\mathbb{A}$) and 163 unknown tokens ($\mathbb{U}$).

Note that whilst sets $\mathbb{W}$, $\mathbb{A}$ and $\mathbb{U}$ are by definition mutually disjoint, the others are not, e.g. some tokens may occur both as cipher and type acronym, or as prefix and unrecognized token (if not in the first position of a name). We define a project's *vocabulary*, for a particular name species, to be the bag of tokens obtained from the multiset union of the 6 sets for that species, e.g. Tomcat's local variable vocabulary consists of 1614 tokens.

Sets $\mathbb{C}$ and $\mathbb{T}$ correspond to removing the duplicate names in $C$ and $T$, because each cipher and type acronym consists of a single token. The names in $P$ consist only of tokens from $\mathbb{R}$ and $\mathbb{W}$, whilst the names in $U$ have at least one component in $\mathbb{A}$ or $\mathbb{U}$, besides possibly from $\mathbb{R}$ and $\mathbb{W}$.

Besides the unique tokens within each set, we also consider all occurrences of tokens in declarations. For Tomcat, the 12 unique ciphers are declared 1417 times, while the 163 unique unknown tokens occur in 743 declarations. Table V shows, for each species, the distribution of each type of token as a proportion of a project's vocabulary and, in parentheses, as a proportion of all occurrences.

TABLE V
Distribution of proportions of unique tokens within vocabulary and, parenthesised, within all occurrences

| | | Field | Argument | Variable |
|---|---|---|---|---|
| $\mathbb{C}$ | **Minimum** | .000(.000) | .003(.001) | .002(.008) |
| | **1st Quartile** | .000(.000) | .009(.034) | .007(.038) |
| | **Median** | .002(.000) | .013(.047) | .009(.050) |
| | **Mean** | .002(.001) | .015(.049) | .011(.061) |
| | **3rd Quartile** | .003(.001) | .019(.064) | .012(.076) |
| | **Maximum** | .009(.008) | .049(.159) | .034(.203) |
| $\mathbb{T}$ | **Minimum** | .000(.000) | .013(.002) | .014(.004) |
| | **1st Quartile** | .003(.000) | .033(.012) | .049(.024) |
| | **Median** | .008(.002) | .052(.023) | .063(.037) |
| | **Mean** | .010(.003) | .053(.028) | .066(.042) |
| | **3rd Quartile** | .014(.004) | .067(.039) | .084(.051) |
| | **Maximum** | .039(.027) | .113(.104) | .123(.157) |
| $\mathbb{R}$ | **Minimum** | .000(.000) | .000(.000) | .001(.000) |
| | **1st Quartile** | .003(.002) | .005(.003) | .005(.007) |
| | **Median** | .005(.007) | .007(.005) | .006(.010) |
| | **Mean** | .005(.021) | .007(.010) | .007(.012) |
| | **3rd Quartile** | .006(.023) | .009(.009) | .008(.015) |
| | **Maximum** | .016(.315) | .021(.081) | .016(.053) |
| $\mathbb{W}$ | **Minimum** | .674(.633) | .653(.202) | .610(.542) |
| | **1st Quartile** | .828(.861) | .763(.768) | .730(.723) |
| | **Median** | .864(.891) | .812(.816) | .773(.775) |
| | **Mean** | .860(.880) | .806(.801) | .766(.767) |
| | **3rd Quartile** | .906(.921) | .850(.860) | .813(.841) |
| | **Maximum** | .963(.981) | .924(.975) | .876(.911) |
| $\mathbb{A}$ | **Minimum** | .008(.013) | .024(.010) | .032(.017) |
| | **1st Quartile** | .033(.040) | .045(.041) | .052(.057) |
| | **Median** | .043(.053) | .059(.063) | .063(.077) |
| | **Mean** | .044(.055) | .057(.076) | .063(.081) |
| | **3rd Quartile** | .054(.068) | .066(.094) | .074(.096) |
| | **Maximum** | .078(.121) | .098(.411) | .100(.393) |
| $\mathbb{U}$ | **Minimum** | .012(.003) | .013(.002) | .028(.008) |
| | **1st Quartile** | .038(.017) | .040(.017) | .059(.025) |
| | **Median** | .061(.037) | .053(.026) | .082(.032) |
| | **Mean** | .080(.040) | .062(.036) | .088(.037) |
| | **3rd Quartile** | .096(.050) | .072(.038) | .101(.044) |
| | **Maximum** | .281(.162) | .227(.370) | .243(.090) |

Table V shows that at least 61% of a project's vocabulary are words, their use being more common in field names. Acronyms and ciphers are most common as formal argument and local variable names. Recognised abbreviations form at most 10% of a project's vocabulary, and the mean and quartile figures are similar for unrecognised tokens ($\mathbb{U}$), but there are some projects where more than 20% of the vocabulary is unrecognised. Outliers are also found in other types of token. The use of redundant prefixes ($\mathbb{R}$) in JUnit, for instance, is an order of magnitude greater than any other project studied, accounting for 31.5% of all tokens in field names. Similarly, the use of words and acronyms as tokens in formal arguments in Groovy is remarkably low at 20.2% of occurrences, some 36% lower than in any other project.

### B. Phrasal Structures (RQ2)

It does not make sense to tag and parse the names in $C$ and $T$, and those in $U$ require further processing for correct tagging and parsing. Thus our second research question only concerns

the 2.6 million declarations in $P$, with names composed of words, acronyms and redundant prefixes.

Applying the Stanford Parser to PoS tagged field names in $P$ identifies the most common phrasal structures shown in Figure 1, where the whiskers extend at most to 1.5 times the interquartile range from the box. Unsurprisingly, given that field names largely represent the attributes of entities and that the JLS and EJS encourage developers to use nouns and noun phrases to name them, the overwhelming majority of field names in $P$ are noun phrases (NP). Redundant prefixes are used in field names in some projects, and these are seen in Figure 1 as a redundant phrase followed by a noun phrase (RDP NP). The lowest outlier for NP and the highest for RDP NP is JUnit, where redundant prefixes are used extensively. The fifth category, a noun phrase followed by a verb phrase (NP VP), contains names that might be a sentence, e.g. `FIELD_IS_VOLATILE` (JDK). An alternative NP VP structure is found in multi-part names such as `ConfigurationView_replaceWith` (Eclipse). We discuss multi-part names further in Section IV. Importantly, Figure 1 shows that though noun phrases are the dominant phrasal structure for field names, other phrasal forms also need to be considered by automated analysis.

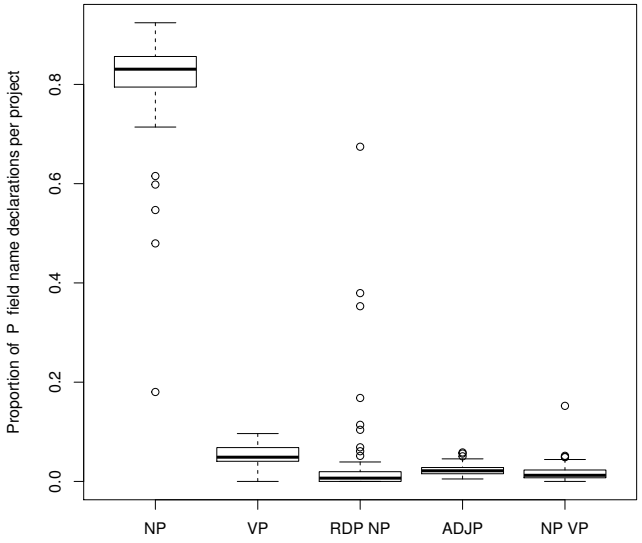



Fig. 1. Proportions of most common field name phrase structures in $P$

Table VI shows the mean proportions of the 5 most common phrase structures for each species. In this and following tables, proportions are given in parentheses when they are not amongst the five most common. The NP VP pattern seen in field names occurs much less often in formal argument and local variable names. Prepositional phrases (PP) are more common in local variable names, e.g. `beforeMethods` (Stripes), and adverbial phrases such as `forward` (OpenProj declaration `boolean forward`) are found in formal arguments.

Non-boolean reference names are predominately noun phrases (Figure 2 and Table VII). The outliers found in JUnit in Figure 1 are also found in Figure 2.

The distribution of phrases in boolean field names is shown in Figure 3. Liblit *et al.*'s observation that developers use

TABLE VI
MEAN PROPORTION OF 5 MOST COMMON PHRASE STRUCTURES IN $P$

| | NP | VP | RDP NP | ADJP | NP VP | PP | ADVP |
|---|---|---|---|---|---|---|---|
| **Field** | .804 | .051 | .039 | .023 | .019 | (.006) | (.005) |
| **Argument** | .908 | .021 | .011 | .022 | (.000) | (.006) | .008 |
| **Variable** | .884 | .023 | .012 | .023 | (.004) | .015 | (.011) |

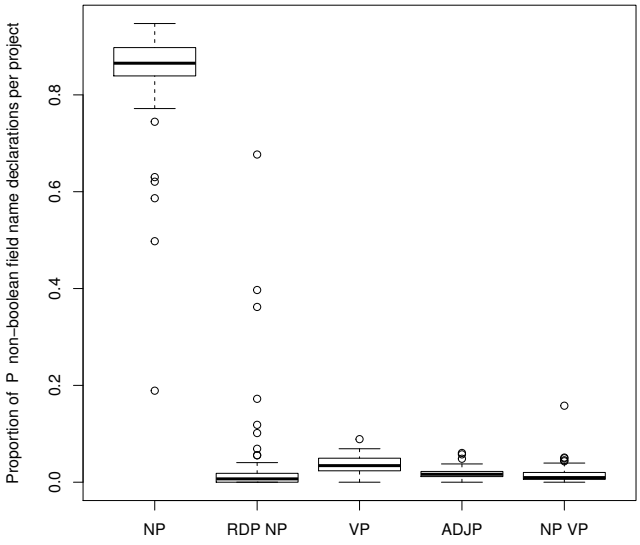



Fig. 2. Proportions of most common non-boolean field name phrase structures in $P$

noun phrases for booleans where the verb 'to be' has been elided may be confirmed by the noun phrase being the largest category. Names beginning with a verb have been divided by the Stanford Parser into two categories, those that are verb phrases (VP) and those that are composed of the 3rd person present form of a verb followed by a noun phrase (VBZ NP), which mixes a PoS tag with a phrasal level tag. The apparently multi-phrase form VP NP arises in boolean names (Table VIII), particularly in local variables, because the Stanford Parser appears to have difficulty parsing some combinations of verbs and nouns. Names like `isShowLines` (JasperReports) are difficult to parse into any form of phrasal structure, because they are not English phrases. We discuss these issues further in Section IV.

In Table VIII some formal argument names are categorised as consisting of adjectives, rather than forming adjectival phrases. The Stanford Parser sometimes categorises names consisting of a single adjective as an adjectival phrase (ADJP) and on other occasions as a sentence fragment containing a single adjective (*JJ*). Testing shows this behaviour to be consistent, and we have left the results as reported by the parser. Summing the figures in the ADJP and *JJ* columns gives the extent of the use of adjectival phrases in boolean names.

Tables VII and VIII answer RQ2 by showing that, for names composed only of prefixes, words and acronyms ($P$),

TABLE VII
MEAN PROPORTION OF 5 MOST COMMON PHRASE STRUCTURES FOR NON-BOOLEAN DECLARATIONS IN $P$

| | NP | RDP NP | VP | NP VP | ADJP | PP | ADVP |
|---|---|---|---|---|---|---|---|
| **Field** | .839 | .040 | .037 | .017 | .019 | (.005) | (.006) |
| **Argument** | .935 | .011 | .008 | (.000) | .018 | (.005) | .007 |
| **Variable** | .902 | .012 | .016 | (.002) | .021 | .015 | (.010) |

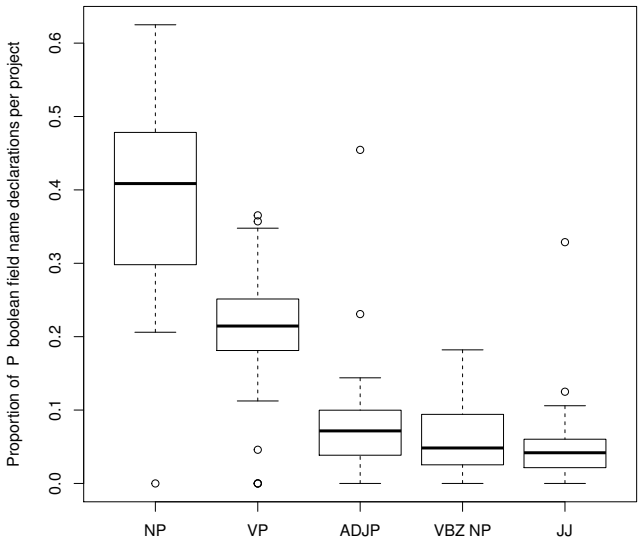



Fig. 3. Proportions of most common boolean field name phrase structures in $P$

developers tend to use the phrasal structures identified by Liblit *et al.*, including the use of nouns or noun phrases as names for booleans, where plausible use of the verb 'to be' has been elided. Table VIII shows use of adjectival phrases (ADJP) and bare adjectives ($JJ$) where, similarly, plausible use of 'to be' has been elided, e.g. `empty` (XOM).

The metaphors proposed by Liblit *et al.* for names [4] were intended to be an approximation to their phrasal structure, rather than a comprehensive list. Two of the metaphors are relevant to Java reference names: *data are things* (which corresponds to the use of noun phrases) and *true/false data are factual assertions* (corresponding to the indicative mood, e.g. `contains`, `isEmpty`). Our results show that some 85% or more of non-boolean field names in $P$, and similar proportions of formal argument and local variable names, are constructed from noun phrases. The finding shows widespread use of Liblit *et al.*'s metaphor and that developers also use other forms of phrasal structure in names.

For boolean names in $P$ the picture is more complex. Liblit *et al.* observed that some boolean names are factual statements with the verb 'to be' elided [4]. The latter case includes isolated nouns, or noun phrases and adjectives, e.g. a local variable named `empty` (XOM), which might easily, and more clearly, be named `isEmpty`. Our results for boolean names (Table VIII) show a large proportion of noun phrases and isolated adjectives. Our manual inspection of isolated adjectives and adjectival phrases confirms Liblit *et al.*'s observation. Many noun phrases could reasonably be preceded by 'is'. Indeed prepositional phrases could also, often, be preceded by 'is'.

From these findings we can identify a production for single phrase reference names: RDP?(NP|VP|ADJP|PP|ADVP). We include the use of redundant prefixes RDP as optional for all single phrases because while they are not always represented in the 5 most common patterns in Tables VI, VII and VIII, reference names such as `fIsDefaultProposal` (Eclipse) and `bInRefresh` (Vuze) are found in practice. The production differs in two regards from the phrase structure grammar used in SWUM [5]: firstly, our production only attempts to describe single phrases, and secondly, as a result of our survey, includes adjectival and adverbial phrases.

TABLE VIII
MEAN PROPORTION OF MOST COMMON PHRASE STRUCTURES OF BOOLEAN NAMES IN $P$

| | NP | VP | ADJP | VBZ NP | VP NP | JJ |
|---|---|---|---|---|---|---|
| **Field** | .394 | .212 | .078 | .062 | (.039) | .044 |
| **Argument** | .408 | .267 | .099 | .057 | (.025) | .052 |
| **Variable** | .431 | .201 | .075 | .072 | .049 | (.047) |

## IV. DISCUSSION

In this section we discuss our results and how they provide opportunities for improving techniques used to analyse names.

### A. Problems for PoS Tagging

We encountered a number of issues that are important for the extension of this work or its application in program comprehension tools, and suggest some potential solutions.

Homographs (words with same spelling but different meaning) can mislead the PoS tagger, e.g. some noun phrases are actually verb phrases where the leading verb has been mistagged as a noun or an adjective. Sometimes this is an error on the part of the tagger, on other occasions information in the name is insufficient to differentiate the use of a word such as 'duplicate' as an adjective, noun or a verb in names like `duplicate_peer_checker` (Vuze). Contextual information from the declaration may support a particular tagging.

Developers do not always use English words in expected ways, though they are used in ways that may be straightforward for humans to understand. An example is `AboutDialog` (OpenProj) where 'about', a preposition, is used as a noun and the name is intended to be a noun phrase. A similar, common unconventional use of words is the specialised use of verbs and verb phrases as nouns in the names of GUI elements that represent user activity, e.g. `SaveAllAction` (Eclipse, NetBeans). In prose, one could use devices such as an article before the phrase, hyphenation, and possibly quotation marks

to indicate the unusual nature of the word usage. However, the last two can't be used in Java names, and the use of articles may be seen as superfluous by developers.

A heuristic may be used to identify the conditions under which a verb might be used as a noun (a grammatical neologism). However, any heuristic would need to be able to recognise the use of verbs in reference names. For example, the exit behaviour of Java Swing top level window classes is controlled by a group of integer constants, including `CLOSE_ON_EXIT`, where the conventional use of the verb is entirely clear and understandable. Similarly, a style of identifier naming using verb phrases for parenthetic pairs of characters or elements may be found in some text processing code. For example, the Eclipse plugin development environment contains code for writing HTML where string constants such as `OPEN_H4` and `CLOSE_H4` are defined. A further consideration is that non-standard use of a word may occur at a very low frequency and, thus, there is limited evidence to support any decision to treat the word differently.

An alternative approach might be to reclassify verb PoS tags as nouns in the names of specific GUI classes, such as `Action`. More detailed study would be required to identify relevant classes for which such re-tagging would be appropriate. However, a hard coded solution may be less desirable than a reliable heuristic.

We also found that some names that might seem easy for humans to identify as phrases are more difficult for software designed to process sentences to interpret. The name `isTopLevelChange` was erroneously tagged as two separate phrases. The issue in this instance is that the parser doesn't see 'top level change' as a noun phrase unless the preceding verb is removed. Through experimentation we found that inserting the determiner 'a' between the verb and the noun phrase would have led the parser to identify the noun phrase as expected.

Another group consists of names composed of English words in non-phrasal combinations, including the nonsensical, such as `ignoreActivate` (Eclipse), `isShowLines` (JasperReports) and `manual_lazy_haves` (Vuze). Names in this group are candidates for refactoring. While it might be straightforward, for example, to refactor `ignoreActivate` to `ignoreActivation`, the refactoring is difficult to justify without source code inspection.

The final group of problematic names to consider in $W$ have more than one phrase. Examples include error reporting values and string constants like `ER_CANT_CREATE_URL` where 'ER' is used as a prefix, and internationalised string constant names that are also used as keys in resource bundles. As keys the names have to contain sufficient information on their purpose to be useful to the reader of the resource bundle. One such constant in Eclipse has the name `CompilersPropertyPage_useprojectsettings_label` and some can be extremely long — 39 tokens in one case. In this case the underscore separates phrases, but concatenating the phrases does not form a single sentence. The concern with this group of names is that mechanisms to parse them using conventional natural language tools may need to make judgements about dividing them into phrases on the basis of typography — which is not used consistently by development teams. In this case, the name represents a string used in the GUI class `CompilersPropertyPage` that is a 'label' widget on the displayed page, and relates to an instruction to use project settings for which the text is available in translation. An alternative solution might be for development teams to use some mechanism to specify their naming conventions so that they might also be used by analytical tools to identify the components of the name.

We also observed what appears to be inconsistent behaviour in the Stanford Parser. Names such as `isZip` (BCEL) are tagged as *is/VBZ zip/NN* and parsed as the phrase tree (SINV (VBZ is) (NP (NN zip))), where SINV is a subject-auxiliary inversion. However, names of the form `isEmpty` (Groovy), tagged as *is/VBZ empty/JJ*, are classified as fragments by the Stanford Parser, with the parse tree (FRAG (VP (VBZ is) (ADJP (JJ empty)))), from which we extract the top-level verb phrase (VP) to categorise the name.

### B. Boolean Names

Two issues appear to contribute to the inaccuracies observed when tagging boolean names. One is where an imperative form of a verb is used at the start of the name that might also be considered a noun, e.g. *request* or *update*. This problem can be attributed to the behaviour of the Stanford PoS tagger in the context of names, where there is so little information to help the tagger differentiate between usage. Consider a name such as `requestValue`: is it a declarative or imperative statement? With only limited information available, the tagger tags 'request' as a noun rather than a verb. This behaviour is quite reasonable, and is a limitation of the PoS tagger in the context of names.

The second issue concerns differentiating between the past participle of a verb and an adjective, for example in a phrase such as 'is enabled'. Santorini [17] provides a series of tests to be applied to whole sentences to distinguish between adjectives and past participles. We have applied Santorini's rules insofar as we could to names in the training and test sets, but the limited information available in names makes it difficult to distinguish the two uses.

A constraint we placed on ourselves was to rely only on the information contained in the name for phrasal analysis, so that we could observe the extent to which developers use Liblit *et al.*'s metaphors. Developers of a tool are able to leverage the declaration context as a source of information to support phrasal analysis. For example, a word such as `request`, in the absence of corroborating evidence such as a subsequent determiner, might be tagged as a noun when it is part of a non-boolean name and otherwise tagged as a noun or as a verb, with the phrase structures resulting from the alternatives used to support a preferred PoS tagging.

### C. Abbreviations and Neologisms

The name partitions $C$ and $T$, containing ciphers and type name acronyms, are used for generic identifiers and contain

no phrasal information. There is little to be gained from expanding ciphers and type name acronyms into words. We found declarations of field names in both partitions in some projects. In most cases, fields with generic names are found in classes with coordinates such as `int x` and `int y`, and in inner classes that implement actions activities such as string processing for the containing class. The use of type acronyms as field names is limited to a few projects. For example JBoss, type acronyms appear to have become part of the project vocabulary for some commonly used classes.

Abbreviations used in identifier names vary from the readily expandable `buf` to mnemonics such as `CSTMBCS` (Derby). The latter are more difficult to expand, though techniques have been proposed [13]. A key problem in abbreviation expansion is that an abbreviation may have more than one expansion. For example, names like `iStream` (NetBeans) and `oStream` (BlueJ) use 'i' and 'o', typically used for 'integer' and 'object', to mean 'input' and 'output'. In these examples, the type name should help identify the correct expansion. Truncations with multiple expansions also cause problems. Among the names in our corpus 'auto' is used as a contraction of 'automated', 'automatic' and 'automatically'. Abbreviation expansion is a complex topic [13][18] and is outside the scope of this paper. Abbreviation expansion could be applied either prior to phrasal analysis or as an iterative solution to identify possible corrections to unanticipated phrase structures. For example, the boolean name `isAutoActivated` (Eclipse) could be expanded to *is/VBZ automated/JJ activated/VBN*, *is/VBZ automatic/JJ activated/VBN* and *is/VBZ automatically/RB activated/VBN*, allowing the latter to be selected as the candidate expansion.

Some truncated abbreviations can be difficult to detect because they are also words (e.g. 'auto' also has the meaning of 'automobile' in American English) and thus give rise to an unintended interpretation of grammatical structure. For example, common abbreviations such as `inFile` (NetBeans) and `outFile` (Ant) are PoS tagged as *IN NN* and thus seen as prepositional phrases by the Stanford Parser.

Other than abbreviations, neologisms and spelling mistakes are included in the $U$ bag. Spelling mistakes can be identified and, potentially, corrected using spell checking software. Techniques exist to identify neologisms derived from existing words, but completely new words and 'grammatical neologisms' are less easy to detect [19].

### D. Research Agenda

We summarise our discussion with a suggested research agenda, listing issues that need to be addressed in order to improve tools that rely on identifier names to support program comprehension, software maintenance and other tasks.

- Improve PoS tagging algorithms in order to:
  - distinguish homographs with different grammatical categories (verb and noun, past participle and adjective, etc.).
  - recognise possessive nouns without apostrophes.
- Develop algorithms to recognise and parse names consisting of multiple phrases.
- Apply neologism recognition techniques to identifiers.
- Develop heuristics to identify non-phrasal combinations of words.

## V. Related Work

In this section we describe two principal themes of related research. The most closely related concerns the investigation and cataloguing of identifier name structure. The second strand concerns the 'pragmatic' grammars used in approaches developed to extract semantic information from identifier names.

Influencing all but the earliest work is Liblit *et al.*'s wide ranging treatise on identifier naming [4], which observed that names are 'pseudo-grammatical utterances' or phrase fragments. On the basis of programming experience and observation, but without systematic quantification of their use in practice, Liblit *et al.* described *metaphors* for identifier names that reflect the role of the name. One metaphor is *data are things*, so that identifiers of data objects are named with nouns or noun phrases, and methods that behave as mathematical functions are named with noun phrases that reflect the returned value [4], e.g. the method `size()` in Java collection classes. Our study provides evidence of the prevalence of phrasal forms matching the use of metaphors in reference names.

The first grammar of identifier name structure was discovered by Caprile and Tonella [20], who analysed C function names. The grammar described the majority of function names. The grammar was subsequently applied to refactor function names to make them more meaningful [21]. A similar analysis of Java method names was undertaken by Høst and Østvold using a specially developed PoS tagger that also considered type names to be a separate part of speech [8], i.e. the PoS tagger also did some semantic processing. Høst and Østvold found a complex grammar with many 'degenerate' forms. They also found a relationship between the structure of a method name and the functionality of the method, sufficient to automate the detection of names that did not accurately describe the implemented methods [22].

We have analysed the structure of Java class names using a model for the Stanford PoS tagger trained on class names [9]. We focused on individual words, and the repetition of tokens in super class and super type names. We found common patterns of PoS tags, but were unable to identify a grammar.

A survey of field names by Binkley *et al.* [23] employed the default Stanford tagger model trained on the Wall Street Journal corpus to analyse C++ and Java field names containing only words, abbreviations and acronyms found in the SCOWL lists up to size 50. The survey used four templates to provide additional context for the PoS tagger, e.g. the tokens were followed by 'is a thing' to nudge the tagger towards treating the name as a noun. Three other templates were used that treated the name as a sentence, a list item, and a verb. The main aim of the survey lay in evaluating the efficacy of the method rather than an exhaustive survey of field name structure. It was found that 88% of names were PoS tagged correctly using the four templates. This is similar to our findings for field names in Table VI for noun and verb phrases. Binkley *et al.* do not

report any use of prepositional phrases, for example, or the long multi-phrase names that we identified.

A survey of class, method and field names in C++ and Java by Gupta *et al.* [7] employs the technique of using WordNet [24] to identify candidate PoS tags that were then used to determine the phrasal structure of names, instead of employing conventional PoS tagging techniques. A simplified set of PoS tags are deemed sufficient for identifiers rather than the Penn Treebank. Gupta *et al.* found that non-boolean field names are typically noun phrases, while boolean fields are generally verb phrases that ask a question. However, their study concerned the evaluation of their PoS tagger and does not quantify their finding on field name structure. Our findings above agree with Gupta *et al.* on the most common structure of non-boolean field names, and offer insights into the other phrasal structures used. However, we contradict their finding on the most common structure of boolean names.

Lawrie *et al.* [10] undertook a statistical investigation of the differences in identifier name quality between 78 proprietary and open source projects written in different programming languages, and projects developed at different times over a 30 year period. Their measures of name quality include the relative proportions of dictionary words, abbreviations, and single-letter abbreviations in names, and the length of names as the number of tokens. Our work differs not just in its intent, but also in the scope and nature of the analysis (Table V). We document the variation in composition of reference names only, rather than all names, and use finer-grained definitions of tokens with a focus on the level of processing the token might require. We give a per-species analysis of name composition to inform the suitable strategies to be adopted by the developers of program comprehension tools .

The literature describes a range of approaches to the extraction of semantic information from names. All rely on PoS tagging to help identify the structure of names. In some cases assumptions are made about the structure of names to simplify processing. The most comprehensive approach so far is adopted in the software word usage model (SWUM) developed by Hill [5]. SWUM uses a general grammar for all species of name to support semantic parsing. The grammar relies on a smaller set of PoS tags than the Penn Treebank and includes productions for noun, preposition and verb phrases. Hill does not quantify name structure, nor the coverage of the grammar for the various species of name. Our work suggests that the grammar used in SWUM should be extended to include adjectival and adverbial phrases.

Abebe and Tonella developed the system of using templates (adopted by Binkley *et al.* [23]) to try to create a statement or sentence that provides additional context to support a PoS tagger [6]. The approach uses `Minipar` to parse the resulting name and template combination, which is rejected if `Minipar` cannot identify an element in the sentence. The approach uses a limited number of templates — 5 for field names — that can only be used to identify a few types of single phrase names. The technique is intended to support concept identification, and was subsequently used to extract ontologies from source code [25]. Richer concept extraction techniques, resulting in more detailed conceptual models, might result from the development of templates for the wider variety of field name structures identified in this paper.

## VI. Concluding Remarks

Reference name declarations constitute around 69% of all declarations in source code and are therefore a rich source of information for developers and tools that perform or support software maintenance tasks, including program comprehension and code search. This paper contributes:

- the first systematic survey of reference names, including the distribution of their components (types of tokens) and forms (phrasal structures);
- the empirical confirmation of extensive adherence to forms suggested in the literature;
- the identification of other reference name forms;
- a research agenda to improve reference name processing.

The survey consists of a quantitative study of the components and most frequent forms of 3.5 million declarations of 522,857 unique field, formal argument and local variable names, extracted from 60 FLOSS Java projects, complemented by an in-depth qualitative observation of individual names, gained from manually tagging almost 46,000 names.

We found that the majority of names use the components suggested in naming conventions (ciphers, type acronyms, dictionary words, etc.) and consist of phrases, especially phrases that largely follow the grammar of Hill and the metaphors observed by Liblit *et al.*, often used by program comprehension tools.

However, our study also found a non-negligible amount of components (e.g. 18% of field names on average) that require further processing (abbreviation expansion or spell and neologism checking), which is a barrier to tool-supported program comprehension. However, we must emphasise that there can be considerable variation in the proportions of the categories between projects with, in extreme cases, more than 70% of the formal arguments in some projects requiring further processing.

Moreover, we found that developers use a richer range of phrases than documented in previous work, including long names composed of multiple phrases, adjectival and adverbial phrases, and non-phrasal names with dictionary words.

All these findings provide insights on how reference names are constructed in practice, and the issues they raise for program comprehension, whether by humans or by software tools that rely on standard techniques. The close inspection of how names are tagged and parsed led us to point to several possible avenues of further research and development in the natural language processing of reference names.

## References

[1] S. Butler, M. Wermelinger, Y. Yu, and H. Sharp, "INVocD: Identifier name vocabulary dataset," in *Proc. of the 10th Working Conf. on Mining Software Repositories*. IEEE, 2013, pp. 405–408.

[2] J. Gosling, B. Joy, G. Steele, G. Bracha, and A. Buckley, *The Java Language Specification (Java SE 8 edition)*, java se 8 ed. Oracle, 2014.

[3] A. Vermeulen, S. W. Ambler, G. Bumgardner, E. Metz, T. Misfeldt, J. Shur, and P. Thompson, *The Elements of Java Style*. Cambridge University Press, 2000.

[4] B. Liblit, A. Begel, and E. Sweetser, "Cognitive perspectives on the role of naming in computer programs," in *Proc. 18th Annual Psychology of Programming Workshop*. Psychology of Programming Interest Group, 2006.

[5] E. Hill, "Integrating natural language and program structure information to improve software search and exploration," Ph.D. dissertation, The University of Delaware, 2010.

[6] S. Abebe and P. Tonella, "Natural language parsing of program element names for concept extraction," in *18th Int'l Conf. on Program Comprehension*. IEEE, Jun. 2010, pp. 156–159.

[7] S. Gupta, S. Malik, L. Pollock, and K. Vijay-Shanker, "Part-of-speech tagging of program identifiers for improved text-based software engineering tool," in *Proc. of the 21st Int'l Conf. on Program Comprehension*, 2013, pp. 3–12.

[8] E. W. Høst and B. M. Østvold, "The Java programmer's phrase book." in *Software Language Engineering*, ser. LNCS, vol. 5452. Springer, 2008, pp. 322–341.

[9] S. Butler, M. Wermelinger, Y. Yu, and H. Sharp, "Mining Java class naming conventions," in *Proc. of the 27th IEEE Int'l Conf. on Software Maintenance*. IEEE, 2011, pp. 93–102.

[10] D. Lawrie, H. Feild, and D. Binkley, "Quantifying identifier quality: an analysis of trends," *Empirical Software Engineering*, vol. 12, no. 4, pp. 359–388, 2007.

[11] S. Butler, M. Wermelinger, Y. Yu, and H. Sharp, "Improving the tokenisation of identifier names," in *25th European Conf. on Object-Oriented Programming*, ser. Lecture Notes in Computer Science, M. Mezini, Ed., vol. 6813. Springer Berlin/Heidelberg, 2011, pp. 130–154.

[12] K. Atkinson, "SCOWL readme," http://wordlist.sourceforge.net/scowl-readme, 2004.

[13] E. Hill, Z. P. Fry, H. Boyd, G. Sridhara, Y. Novikova, L. Pollock, and K. Vijay-Shanker, "AMAP: Automatically mining abbreviation expansions in programs to enhance software maintenance tools," in *Proc. of the 5th Int'l Working Conf. on Mining Software Repositories*. ACM, 2008, pp. 79–88.

[14] K. Toutanova, D. Klein, C. Manning, and Y. Singer, "Feature-rich part-of-speech tagging with a cyclic dependency network," in *Proceedings of HLT-NAACL*, 2003, pp. 252–259.

[15] M. Heller and C. Simonyi, "The Hungarian revolution," *BYTE*, vol. 16, no. 8, pp. 131–138, 1991.

[16] D. Klein and C. D. Manning, "Fast exact inference with a factored model for natural language parsing," in *Advances in Neural Information Processing Systems 15*, 2002, pp. 3–10.

[17] B. Santorini, "Part-of-speech tagging guidelines for the Penn Treebank Project," Department of Computer and Information Science, University of Pennsylvania, Tech. Rep. MS-CIS-90-47, 1990. [Online]. Available: http://repository.upenn.edu/cis_reports/570/

[18] D. Lawrie, D. Binkley, and C. Morrell, "Normalizing source code vocabulary," in *Proc. of the International Working Conference on Reverse Engineering*, 2010.

[19] M. Janssen, "Neotag: a pos tagger for grammatical neologism detection," in *Proceedings of the Eight International Conference on Language Resources and Evaluation (LREC'12)*. European Language Resources Association (ELRA), 2012.

[20] B. Caprile and P. Tonella, "Nomen est omen: analyzing the language of function identifiers," in *Proc. Sixth Working Conf. on Reverse Engineering*. IEEE, Oct 1999, pp. 112–122.

[21] ——, "Restructuring program identifier names," in *Proc. Int'l Conf. on Software Maintenance*. IEEE, 2000, pp. 97–107.

[22] E. W. Høst and B. M. Østvold, "Debugging method names," in *Proc. of the 23rd European Conf. on Object-Oriented Programming*. Springer-Verlag, 2009, pp. 294–317.

[23] D. Binkley, M. Hearn, and D. Lawrie, "Improving identifier informativeness using part of speech information," in *Proc. of the Working Conf. on Mining Software Repositories*, 2011.

[24] C. Fellbaum, Ed., *WordNet: an electronic lexical database*. Bradford Books, 1998.

[25] S. L. Abebe and P. Tonella, "Towards the extraction of domain concepts from the identifiers," in *Proc. of the 18th Working Conf. on Reverse Engineering*. IEEE Computer Society, Oct. 2011, pp. 77–86.