Investigating Naming Convention Adherence in Java References

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

Abstract—**Naming conventions can help the readability and comprehension of code, and thus the onboarding of new developers. Conventions also provide cues that help developers and tools extract information from identifier names to support software maintenance. Tools exist to automatically check naming conventions but they are often limited to simple checks, e.g. regarding typography. The adherence to more elaborate conventions, such as the use of noun and verbal phrases in names, is not checked.**

We present NOMINAL, a naming convention checking library for Java that allows the declarative specification of conventions regarding typography and the use of abbreviations and phrases. To test NOMINAL, and to investigate the extent to which developers follow conventions, we extract 3.5 million reference — field, formal argument and local variable — name declarations from 60 FLOSS projects and determine their adherence to two well-known Java naming convention guidelines that give developers scope to choose a variety of forms of name, and sometimes offer conflicting advice. We found developers largely follow naming conventions, but adherence to specific conventions varies widely.

I. INTRODUCTION

Identifier naming conventions [1][2] are intended to support the readability of source code and also provide some cues, such as typography and suggestions for phrase structure and the use of words and abbreviations, that facilitate the extraction of meaningful information. Where developers follow conventions there is less work required for program comprehension tools to split names [3][4] and extract semantic content from source code [5][6]. Besides naming conventions, tools that extract semantic content also rely on observations of naming practice [7] to support their approach to phrasal analysis.

In this paper we investigate the adherence to naming conventions for Java references — fields, formal arguments and local variables — for two important reasons. Firstly, reference names constitute 52% of the unique names and around 70% of the declarations in a corpus of 60 FLOSS Java projects [8], making them a potentially large source of information for program comprehension and software maintenance tools. Secondly, unlike classes and methods where naming conventions are largely clear and consistent [9][10], reference naming conventions often provide conflicting advice, e.g. to use abbreviations for formal arguments [1] and not [2], and provide a bewildering choice of forms, including ciphers (single letter abbreviations in conventional settings, e.g. `i` as an integer loop control variable), type name acronyms, abbreviations and phrases. Consequently, software development tools that use the textual and natural language features of identifiers are processing an information source of unpredictable quality.

To check the adherence to such diverse naming guidelines we developed our own convention checker, because available tools such as CheckStyle [11], Google CodePro AnalytiX[1] and PMD [12], do not handle phrasal structures. To ease the check of different selections of rules, we also defined a domain-specific language to define conventions in a simple and declarative way. The language and the checker, called NOMINAL, were informed by our previous study of Java reference names [13] that observed the forms reference names take in practice without *a priori* judgement about which forms are 'right'. In this paper, we explicitly judge names as conventional or unconventional, depending on their adherence or not to a naming guideline. Moreover, this paper looks at typography, which was not covered by our previous study.

To evaluate the effectiveness and efficiency of the proposed approach, we defined three different sets of comprehensive naming conventions with NOMINAL and checked the adherence of 3.5 million reference name declarations against each guideline. The naming conventions are taken from the "Java Language Specification" [1], the "Elements of Java Style" [2], and studies of Java identifiers in the academic literature [5][6][7][13].

In developing and applying NOMINAL to check adherence to reference name conventions, we seek to answer the following research questions:

- **RQ1** To what extent do projects adhere to particular naming conventions or style?
- **RQ2** Do some naming conventions tend to be violated more frequently than others?

The remainder of the paper is organised as follows. In Section II we describe our methodology and give an overview of the naming conventions checked. In Section III we explore the difficulties of testing individual conventions. We address the two research questions and report our results respectively in Sections IV and V. In Section VI we give an account of related work, and draw conclusions in Section VII.

II. METHODOLOGY

In this study we investigate the adherence of reference names to naming conventions in 60 FLOSS Java projects developed in English. We investigate only Java source code

[1] https://developers.google.com/java-dev-tools/codepro/doc

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 check the boolean naming metaphors of Liblit *et al.* [7]. 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 [8], a freely available database of all names occurring in 60 well-known Java FLOSS projects[2]. Among other information, INVocD records for each declared name its *species* — i.e. class, method, field, etc. — and its type.

The corpus for this study is the bag (multiset) of all reference name declarations in INVocD: 626,262 fields, 1,556,931 formal arguments, and 1,319,071 local variables. We check declarations instead of just the names because the applicable convention depends on the name's usage context including species and type.

Each declaration indicates the need for developers to choose a name. If the developer reuses a particular name, it reflects the preference for certain identifier forms, typographical styles, phrasal structures or metaphors. The reuse of names is indeed substantial. The 3,502,264 declarations introduce only 272,228 unique field names, 81,201 unique formal argument names and 169,428 unique local variable names. Reuse rates in the corpus are thus 2.3, 19.2 and 7.8 times for field, formal argument and local variable names respectively.

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% adherence to conventions (namely `int i`), whereas considering only unique names or unique declarations would lead to a distorted figure of 50% adherence, when in fact the developer made 100 choices, only one of which deviated from naming guidelines.

We use the term *token* for a component of a name, whether it is a dictionary word, an acronym or an abbreviation.

B. Naming Conventions

We selected three sets of naming conventions to test names against. The first two are described in the "Java Language Specification" [1] and "The Elements of Java Style" [2]. We refer to these naming conventions as JLS and EJS respectively.

JLS defines both typography — the use of upper and lower case letters and separator characters such as underscores — and content for Java identifier names. The JLS naming conventions remain largely unchanged since initial publication in 1996 and are familiar to most Java developers. JLS defines a simple typography scheme for reference names. Constant field names — those declared with both the `final` and `static` modifiers — are in upper case with the underscore used to separate tokens, e.g. `LEAF_IMAGE` (from Google Web Toolkit). All other field names, and formal argument and local variable names begin with a lower case letter, and the first letter of each subsequent word is capitalised (sometimes known as camel case), e.g. `oldValue` (JabRef).

EJS began life as an internal Java style guide at Rogue Wave Software, and reflects the practice of the company's Java developers. EJS consists of a group of general naming conventions (Rules 9–14) that provide general advice on identifier names — that they should be meaningful, for example — and conventions that provide advice on the typography and content of each species of name (Rules 25–31 cover field, formal argument and local variable names). The typography is, with the exception of acronyms, identical to that defined in JLS. While JLS permits a wide range of content in names, EJS expresses a preference for the use of dictionary words in names, with a few exceptions.

A survey of naming practices by Liblit *et al.* [7] identified a number of *metaphors*, phrasal forms that reflect the role of an identifier name. For example, "true/false data are factual assertions". Liblit *et al.*'s observations both increase the variety of phrasal forms expected, and require the division of species into boolean and non-boolean subspecies. Liblit *et al.*'s metaphors have been used as a starting point for the extraction of information from names [5][6] and our previous work confirms the metaphors are widely used [13]. Further, recent research [7][5][6][14][13][9] shows that developers use a range of phrasal structures wider than specified in EJS and JLS.

We therefore defined a third set of naming conventions that amalgamates the EJS conventions, because of their emphasis on the use of dictionary words, with the observations of phrasal structures found in practice made by Liblit *et al.* [7], Hill [5], Abebe and Tonella [6], Binkley *et al.* [14], and ourselves [13]. We refer to this latter group as AJC, for **a**ggregated **J**ava **c**onventions. By evaluating adherence to AJC we hope to understand the extent to which the observed diversity of phrasal structure is used in accordance with perceived conventions.

C. `Nominal`

To support the evaluation of adherence to naming conventions we developed NOMINAL, a freely available Java library that allows the declarative specification of naming conventions[3]. Each set of conventions — EJS, JLS and AJC — is defined in a separate configuration file.

NOMINAL consists of two components: an evaluation engine that determines the adherence of a name to naming conventions, and a configuration language that allows the declaration of rules read by the evaluation engine.

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

[3] https://github.com/sjbutler/nominal

EJS and JLS specify the identifier species or subspecies (e.g. constant field) to which a convention applies. The Nominal configuration language follows this pattern, defining a range of species and subspecies for which rules may be declared. For each species or subspecies a set of rules may be given that specify the typography, content and other characteristics of a declaration. For example, Figure 1 shows the definition of rules for the *local-variable* species and *local-variable-boolean* subspecies for the AJC conventions. The labels outside the block are hierarchical with the species name at the left and subspecies names following. The labels form trees with a single species at the top of each tree.

```
local-variable {
    first-char: lower;
    body: mixed;
    content: cipher, NP;
}

local-variable-boolean {
    content: cipher,NP,VP,AdjP,AdvP,PP;
}
```

Fig. 1. Nominal rule definitions for AJC showing rule inheritance and overriding

To avoid repetition or verbose rules, Nominal follows a simple model of rule inheritance. The rule for 'local-variable-boolean' inherits the typographic rules from 'local-variable'. The content field overrides the definition in the parent species and allows the use of noun phrases and verb phrases[4], observed by Liblit *et al.* [7], and of adjectival, adverbial and prepositional phrases, that we observed [13]. Ciphers are the only non-phrasal form of names permitted.

Nominal also defines some default rules for each species/subspecies hierarchy including the use of separator characters (none), the use of plurals (unspecified) and redundant prefixes (none). Typically, these rules are used in only a few subspecies.

The input to Nominal is a name with metadata about the declaration context (including its type, species and any modifiers used in the declaration) and tokenised versions of the name, as well as part of speech (PoS) tags, and its phrasal structure. This allows users to use any tokeniser, tagger and parser of their choice. For this study, we tokenised the names with INTT [3] and used the PoS tagger for reference names we developed in previous work [13]. The PoS tagger has an accuracy of 85% for reference names and 95% for individual tokens. The Stanford Parser [15] was used to identify the name's phrasal structure using the technique we applied in previous work [13].

The evaluation engine contains objects that evaluate each rule specified in the configuration file. There is, for example, an object that evaluates the typographical rule for the initial character of a name, and another that evaluates the use of separator characters. The results for each rule are recorded in an information object that annotates the name declaration passed to the evaluation engine. When all the rules have been evaluated, the name declaration object is returned to the caller. The approach allows rules to be evaluated individually using simple approaches.

D. Threats to Validity

There are at least three threats to *construct validity*. First, while the conventions used are sufficiently generic and well-known that they are likely to have been followed, there may also have been project specific naming conventions in place that we have not captured and thus cannot test against.

Second, whilst the accuracy of the PoS tagger, trained on 30,000 unique field names [13] from the INVocD corpus, is high, it is not perfect.

Third, phrase structure grammars are context free and recognise the aggregation of tagged words into grammatically coherent groups, but there is no guarantee that they are meaningful. For example, 'The mat sat on the cat' is grammatically correct, but absurd. Accordingly, the threat arises from an underlying assumption that developers have created meaningful rather than absurd names.

Threats to *external validity* arise because we constrained our experiment in two dimensions. First, we analysed only projects developed in Java, prior to edition 8, to take advantage of its strong typing; and, secondly, we analysed projects where names were constructed using English words. Accordingly we cannot be sure whether our findings may be applied to programming languages with weaker typing, or whether developers creating names using languages other than English use similar phrasal structures.

III. Checking Naming Conventions

In this section we explore which naming conventions can be reliably and accurately checked. The intention of the authors of the EJS and JLS conventions was to provide guidance to developers. Consequently, convention definitions sometimes lack the precision required to make them easy to test. For example, JLS suggests the use of *mnemonic terms* in local variable and formal argument names similar to those used as "parameters to widely used classes." [1] The few examples given reflect the use of names taken from Java library classes, but they are not the only widely used classes, particularly in teams using specific library APIs, making the distinction between well-known classes and others arbitrary, and, thus, difficult to test. Reliable checking of conventions requires a convention to be defined with a clear statement of which declarations it applies to and readily identifiable boundaries between declarations that conform to the convention and those that do not. Categorising a declaration is straightforward where a convention is applied to a species. However, where the convention is defined for a subspecies, additional distinguishing information is required to categorise a declaration, e.g identifying a loop control variable

[4] Nominal uses Penn Treebank notation for phrases: ftp://ftp.cis.upenn.edu/pub/treebank/doc/tagguide.ps.gz

declaration can require more information than is contained in the declaration itself.

A. Name Content Conventions

EJS and JLS specify a range of content for identifier name tokens. There are differences in the token content specified by the two conventions. EJS advocates the use of dictionary words (Rules 9 and 12), for example, while JLS expresses a preference for the use of abbreviations for formal argument and local variable names. Between them, EJS and JLS define five types of identifier name token content:

1) **Cipher**: JLS specifies a largely familiar set of single letter names (Table I) to be used "for temporary and looping variables, or where a variable holds an undistinguished value of a type."
2) **Type Acronym**: specified in JLS for short-lived formal argument and local variable names, type acronyms are single token names that are acronyms of the declared type, e.g. `StringBuilder sb`.
3) **Acronym**: either a pronounceable acronym such as AWOL, or an initialism like XML.
4) **Word**: a word found in a dictionary.
5) **Abbreviation**: a string of letters and, possibly, digits, that does not match any of the preceding four categories.

Tokens are tested and annotated with one or more objects indicating which of the five categories they belong to. The first test determines whether the name consists of a single token and if that token is a cipher. A recognised cipher is further tested to determine if it is of the correct type. A name that is not a correct used cipher, if a single token, is then tested to determine if it is a type acronym. For example, a declaration of `Iterator i` would fail the test for a correctly used cipher (using the list of ciphers in Table I) and would be classified as a type acronym. Single tokens failing the first two tests and each of the tokens found in longer names, are tested using MDSC to determine whether they are words or acronyms. Where a token is not recognised, it is categorised as an *unrecognised* abbreviation.

MDSC[5], a freely available multi-dictionary spell checking library for identifier names, contains lists of abbreviations, acronyms and words from the SCOWL word lists [16] with additional lists of technical terms, abbreviations and acronyms taken from our own work [3][9] and the AMAP project[6] [17].

EJS (Rule 28) states that developers should compile their own list of acceptable ciphers and 'shorthands' and offers a list of suggestions, a few of which (`c`, `e`, `o` and `s`) are also specified by JLS. EJS also states that using the list given in JLS is acceptable. We amalgamate the JLS list of ciphers with the EJS suggestions to enable us to identify commonly used ciphers when testing the EJS conventions, rather than mark all but a few ciphers as incorrect. EJS also suggests the use of the ciphers `x`, `y` and `z` for coordinates. As there is no direct type correspondence we widen the definition of a coordinate to include any numeric type.

TABLE I
JLS CIPHERS AND THEIR CORRESPONDING TYPES

Cipher(s)	Type(s)
b	byte
c	char
e	Exception
d	double
f	float
i,j,k	int
l	long
o	Object
s	String
v	a value of some type

Identifier name content is specified in NOMINAL rules by a line starting with the keyword `content` followed by one or more content types, including `cipher`, `type-acronym`, `abbreviation`, and phrasal structure expressed using the Penn Treebank phrase name abbreviations AdjP, AdvP, NP, PP and VP. For example the content of a local variable in JLS is specified as `content: cipher,type-acronym,abbreviation,NP`. The first three content types are relevant to single token names only, with `abbreviation` indicating that standalone abbreviations such as `buf`, specified in JLS, are accepted. The phrase name abbreviations can be used alone to specify single phrases, or combined to specify more complex phrase structures. For example, NP VP specifies a noun phrase followed by a verb phrase.

AJC follows the same typography scheme as EJS and divides declarations into finer-grained subspecies than EJS and JLS. Rather than, for example, treating all local variable declarations identically, names declared boolean can have a phrase structure that includes the use of verb and adjectival phrases (Figure 1). AJC also includes subspecies specifying phrasal structures for string constant declarations and references to GUI actions, where we have observed the use of alternative phrase structures [13].

B. Typographical Conventions

Typographical conventions are clearly defined in JLS and EJS except for two grey areas: the use of single underscores in constant field names in JLS, which was resolved by implication from the examples, and the use of upper case acronyms in JLS, which we left unchecked. EJS is unequivocal in stating that only the first letter of an acronym is capitalised when appropriate (Rule 13). In contrast, JLS makes no comment on the matter, but quotes examples of camel case names with upper case acronyms, e.g. the method name `toGMTString()`, which would be `toGmtString()` according to EJS.

The rules we applied for all three conventions are that names declared as constant fields are upper case with a single underscore character between each token. Other names are composed of mixed case or camel case, that is the first letter is lower case, the initial letter (if there is one) of each subsequent

[5] https://github.com/sjbutler/mdsc

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

token is capitalised and the remainder are lower case for EJS (for JLS recognised acronyms may also be upper case). Prior to evaluating the typography, each token is categorised as a word, acronym or abbreviation using MDSC, allowing the application of different typographical rules for acronyms for EJS.

Testing the typographical conventions for constants and variables is undertaken by checking the appropriate use of underscores, the capitalisation of the first character of the name, and the capitalisation of the remaining tokens.

We test the typography of individual tokens rather than the entire name for two reasons. First, to evaluate context based typographical rules for acronyms. Second, the typographical boundaries marked by the developer do not always match the boundaries between tokens, and token boundaries are sometimes omitted, for example `MAXOPEN_DEFAULT` (AspectJ) consists of the tokens {`MAX`,`OPEN`,`DEFAULT`}, so should be `MAX_OPEN_DEFAULT` to comply with the typography rule for constants. This approach offers significant advantages over the regular expressions used by CheckStyle and PMD. Chiefly, checks at the token level determine that tokens have the correct typography, rather than the name having the appearance of being typographically correct, which is what a regular expression can test. NOMINAL also allows the specification of different typographical rules for acronyms from those applied to words.

C. Reference Naming Conventions Tested

1) Field Names: JLS conventions for field names are expressed in two NOMINAL rules *field-constant* and *field-variable*. The typography is as described previously. JLS specifies the content of constant fields as one or more words, abbreviations or acronyms, and of an "appropriate" part of speech. Given the specification of more than one token we have assumed that any appropriate phrase is permitted.

The EJS field name conventions are expressed in four subspecies rules: *field-constant*, *field-constant-collection-reference*, *field-variable* and *field-variable-collection-reference*. The collection conventions will be explained further below. The typography rules are identical to JLS, with the exception of acronyms in variable names noted above. The content, however, is restricted to noun phrases for constants, and ciphers and noun phrases for variables reflecting EJS's preference for the use of dictionary words (Rule 9) and the specification of nouns/noun phrases.

AJC adds further subspecies to *field-constant* and *field-variable*. *field-constant-string* contains the specification of a 'complex' phrase type in the content rule to include the very long string constants observed in previous work [13]. A further subspecies with the suffix *-action* is used isolate references to instances of classes representing GUI actions that are sometimes named with what appear to be verb phrases, e.g. `SaveAction` (Freemind). GUI actions are defined as declarations of implementations of the Swing `Action` interface and subclasses of `java.util.EventObject`, the superclass of AWT and Swing events.

2) Formal Argument and Local Variable Names: The typographical rules for formal argument and local variable names are identical for JLS, EJS and AJC, with the exception of acronyms. Again EJS does not allow the use of type acronyms or abbreviations, and limits content to ciphers and noun phrases. AJC, as with variable field declarations, defines subspecies for boolean names and references to GUI actions, and permits the use of a wider range of phrases, though limited to particular subspecies.

3) Collections: EJS (Rule 27) specifies that the names of collection references should be pluralised. EJS gives the examples of array declarations — including

```
    Customer[] customers
```

— which are trivial to identify, and the use of collections classes. We constrain EJS's definition of collection class to those classes in the `java.util` package that implement the `Collection` interface. Note that the `Map` classes are not considered collections by this definition. We also apply the rule to collections declared with generic types, which were not part of the Java language when EJS was written. For example, the declaration `List<Customer> customers`.

The idea of pluralising collection names appears reasonable. However, the EJS rule is very narrowly defined permitting only the use of plurals in generic names where the name is the plural of the type. However, we have seen wider use of plurals, such as `List lines` (Antlr and many others), and in arrays of primitive types, e.g. `String[] lines` (Rapla). To this end we check both the occurrence of EJS pluralisation, and pluralisation of the names of collections and arrays in order to understand the extent to which developers use plurals in declarations. The EJS rule for pluralisation defines the names it applies to in a way that makes it difficult for the current version of NOMINAL to express the rule. Accordingly the test for EJS pluralisation is hardwired in the library. The more generic rule used to evaluate more widespread use of plurals in reference collections is specified in AJC as follows:

```
local-variable-collection-reference {
    content: cipher,NP;
    plural: true;
}
```

Although it is implied that non-collection names should not be plurals, EJS does not make an explicit statement to that effect. Accordingly, we treat the pluralisation of names of non-collection references as *unspecified* for the EJS conventions. However, in the AJC conventions the rules for all non-collection references are specified as *singular* to help understand whether declarations that might be expected to be singular are being pluralised.

D. Other Conventions

In addition to testing for the reference name conventions defined in EJS and JLS, each declaration is tested by default for other known conventions such as the use of leading underscores and redundant prefixes. While contrary to the conventions specified by EJS and JLS, it is helpful to understand if non-conformance with EJS and JLS results from developers following other conventions.

Some developers use redundant prefixes in reference names, either indicating the role of the name or the type, for example the field name `mPropertyName` (FreeMind) where `m` represents 'member'. Neither EJS, JLS nor AJC specify the use of redundant prefixes, so any prefix use will be seen as unconventional by NOMINAL. Every multi-token name is tested to determine if the first token is a role-based prefix (`f` for field, `m` for member and `p` for parameter), or a prefix representing a Java primitive type (i.e. `b`, `c`, `d`, `f`, `i` and `l`). The use of a redundant prefix is recorded, and we also record if it reflects the expected role or type.

E. Conventions not Fully Tested

Some rules specified in EJS and JLS cannot be fully tested. EJS Rule 9 specifies the use of 'meaningful names' and is tested partially by ensuring names are constructed of dictionary words, but does not test whether the name is 'meaningful'. Rule 10 suggests the use of 'familiar names', which the examples imply means business domain terms. Rule 10 might be policed in practice with a dictionary of project-approved domain terms, but in a survey of multiple projects in diverse domains the task is not practical. Rule 11 alerts developers to potential design issues by asking them to 'question excessively long names'. We do not evaluate Rule 11 because objective criteria are difficult to identify. Rule 14 specifies that names in the same scope should not differ by case alone, and was tested to the extent that at least one of the names would have unconventional typography. Rules 29 and 30 concern the use of the `this` keyword to distinguish field names, and that arguments to constructors and mutator methods should have the same name as the fields they are assigned to. Neither rule was tested as they apply to the context the name is used in, rather than its structure and content.

JLS lists a number of standard abbreviations for generic formal argument and local variable names, such as `buf` for a buffer. However, only a few illustrations are given as guidance. We test for the use of abbreviations on the assumption that the lists of abbreviations we use, taken from identifier names and in dictionaries, reflect the intent of JLS.

To summarise, we found that the majority of conventions specified in JLS can be evaluated accurately as they are stated. The exception is the vague definition of 'mnemonic' terms. EJS is less sharply defined and in we needed to make assumptions to define a clear boundary to some rules. Three EJS rules (25, 26, and 31 concerning typography of references, and phrasal content) are implemented as stated. Seven further rules required interpretation or are partially implemented. Four rules were ignored because they require project specific information or relate to the way in which the name is used.

IV. ADHERENCE TO SPECIFIC CONVENTIONS (RQ1)

RQ1 concerns the extent to which projects adhere to each convention.

A. Typography

The typographical conventions used in all three sets of naming conventions checked differ only on the typography of acronyms. Across the 60 projects, developers follow conventional typography much of the time (Table II gives figures for JLS where the typography of acronyms is ignored). Acronyms are found, on average, in 10% of field declarations, 7% of formal arguments and 9% of local variables. Adherence to the EJS scheme of mixed case acronyms is found at high levels in most projects, but a minority of projects use upper case acronyms extensively.

There is some variation in typographical conformance between projects that may indicate the use of project specific conventions or a sub-optimal application of typographical conventions. The lowest conformance with typographical conventions occurs amongst field declarations. Inspection of collected declarations suggests that a contributing factor may be developers not following the conventions on when to use constant notation, and, perhaps, the declaration of a field having been changed from constant to variable, or vice versa, without the typography being revised. In Beanshell, for example, there are field declarations such as `final static int normal` and `private static Object NOVALUE`.

TABLE II
DISTRIBUTION OF THE PERCENTAGE OF DECLARATIONS ADHERING TO TYPOGRAPHY CONVENTIONS

	Field	**Formal Argument**	**Local Variable**
Minimum	.43	.80	.78
1st Quartile	.74	.95	.92
Median	.82	.97	.96
Mean	.80	.96	.94
3rd Quartile	.88	.99	.97
Maximum	.97	1.0	1.0

Another possibility is that the definition of a constant used by some developers is more nuanced than that given in JLS or EJS. The Google Java Style guide[7] argues that only those reference declarations that hold a single immutable value should have the typography of a constant. For example, while a declaration of an instance of a collection class using the modifiers `static` and `final` is constant in the sense that the reference to the collection does not change, the values or references stored in the collection can change, so the name should not have the typography of a constant. In contrast, an instance of `java.lang.String` declared `static` and `final` is immutable and therefore constant, and should have the appropriate typography.

A source of noise in some projects is the `java.io.Serializable` interface where each implementing class declares and assigns a value to the field `private final static long serialVersionUID`. The typography is not that of a constant and is imposed by the Java library. Indeed, our analysis shows that the Java Library is just above the mean level of adherence to the typographical and content conventions in JLS.

[7] https://google-styleguide.googlecode.com/svn/trunk/javaguide.html#s5-naming

Eclipse appears to have its own field naming conventions, which differ from those we tested against. Some constant fields are named using conventional upper case typography (e.g. `VISIBILITY_PREF`). Field declarations that reference strings containing UI messages are not declared as constants, and have a structure in which the first element is the name of the class the message originates from followed by either the message, or an explanation and a message (e.g. `LaunchView_Error_1`). In addition, many variable field declarations have redundant prefixes (e.g. `String fPrefillExp`).

The lowest proportion of compliant field declarations is found in MPXJ where variable declarations have the prefix `m_`. Jetty, the next lowest at 52%, prefixes some field name declarations — both constant and variable — with one or more underscores, e.g. `_dynamic`. The least compliant typography for formal argument declarations is found in Vuze where a leading underscore is used for some declarations to distinguish them from field declarations when used to pass values or references to fields in mutator methods, and use of the C style underscore separator in some formal declarations, e.g. `new_value`. Vuze also has the least compliant typography of local variable declarations (78%). As with formal arguments, underscores are used both as separators and prefixes.

B. Name Content

Of the conventions tested, JLS specifies the widest range of content, while EJS is more restrictive. AJC permits a wider range of phrasal content, but only for particular subspecies. Table III shows the distribution of proportion of declarations in the projects investigated that conform to the content conventions for each set of conventions tested. In several cases, differences between EJS and AJC are only apparent at 4 or 5 significant figures.

TABLE III
Distribution of declarations adhering to content rules of each convention. Parenthesised figures include declarations with redundant prefixes

		Field	**Parameter**	**Local**
JLS	**Minimum**	.27(.77)	.84(.90)	.87(.90)
	1st Quartile	.83(.87)	.93(.94)	.92(.93)
	Median	.87(.89)	.94(.95)	.92(.94)
	Mean	.85(.89)	.94(.95)	.92(.94)
	3rd Quartile	.90(.91)	.95(.96)	.93(.95)
	Maximum	.94(.96)	.99(.99)	.97(.97)
EJS	**Minimum**	.27(.75)	.79(.80)	.70(.72)
	1st Quartile	.82(.86)	.88(.89)	.84(.86)
	Median	.86(.87)	.90(.91)	.87(.88)
	Mean	.83(.87)	.90(.91)	.86(.88)
	3rd Quartile	.89(.90)	.93(.93)	.88(.90)
	Maximum	.93(.96)	.97(.97)	.96(.96)
AJC	**Minimum**	.27(.76)	.75(.75)	.70(.72)
	1st Quartile	.86(.89)	.85(.85)	.84(.86)
	Median	.90(.91)	.87(.88)	.87(.88)
	Mean	.87(.91)	.87(.88)	.86(.88)
	3rd Quartile	.92(.93)	.90(.90)	.88(.90)
	Maximum	.97(.97)	.97(.99)	.96(.96)

The differences between the parenthesised and unparenthesised figures illustrate the wide differences in the use of redundant prefixes in the projects studied. We discuss this further below. The concept of correct content for formal arguments and local variables in JLS includes the use of type acronyms, which are excluded by EJS and AJC. Table IV shows the distribution of type acronyms in declarations in the 60 projects surveyed. All maximum values occur in the same project, Polyglot, where 4.5% of fields, 13% of formal arguments and 21% of local variables are type acronyms.

TABLE IV
Distribution of the usage of type acronyms in name declarations

	Field	**Parameter**	**Local**
Minimum	.00	.00	.01
1st Quartile	.00	.02	.04
Median	.01	.03	.06
Mean	.01	.04	.06
3rd Quartile	.01	.06	.08
Maximum	.05	.13	.21

C. Conventional usage of phrases

Table V shows the distribution of the use of conventional phrases in name declarations with phrasal content (i.e. not ciphers and type acronyms) in the 60 projects investigated. The declarations surveyed include the use of redundant prefixes, which are removed from the name before the remainder is PoS tagged.

TABLE V
Distribution of the proportions of declarations with expected phrase structures

		Field	**Parameter**	**Local**
JLS	**Minimum**	.77	.75	.61
	1st Quartile	.87	.85	.77
	Median	.89	.88	.80
	Mean	.89	.88	.79
	3rd Quartile	.91	.90	.83
	Maximum	.96	.97	.91
EJS	**Minimum**	.76	.72	.60
	1st Quartile	.86	.84	.77
	Median	.87	.88	.80
	Mean	.87	.87	.79
	3rd Quartile	.90	.90	.83
	Maximum	.96	.97	.91
AJC	**Minimum**	.76	.75	.61
	1st Quartile	.89	.85	.77
	Median	.91	.88	.80
	Mean	.91	.88	.79
	3rd Quartile	.93	.90	.83
	Maximum	.97	.97	.91

There are only minor differences between the distribution of the proportions of conventional phrase use in declarations for each of the three conventions. The phrasal rules for AJC are applied at a finer granularity of subspecies and allow a wider range of phrase structures than EJS and JLS. Sensitivity to a wider range of phrases makes a difference for some projects, but, generally, the improvement is marginal. This suggests that while developers do use a richer selection of phrases than advocated in EJS and JLS, they do so with a relatively small proportion of declarations.

Plurals following the narrow EJS convention are extremely rare in our corpus. Developers pluralise collection and array references, but not consistently. They also pluralise some references to numeric values and non-collections objects.

In summary, there is no simple answer to RQ1. There is considerable variation in the extent to which the projects surveyed follow typographical conventions. In particular, developers in some projects appear to have difficulty adhering to the conventions for field declarations. Developers tend to follow the JLS conventions on the content of name declarations rather than EJS, with many projects using type acronyms. The phrasal content of names follows conventions closely in most projects, though local variable declarations are less compliant than field and formal argument declarations.

V. Commonly Broken Conventions (RQ2)

RQ2 asks which naming conventions are most commonly broken or ignored.

A. *Typography*

The use of leading underscores in declarations is found in just under half of the projects investigated. The greatest use is found in field name declarations in Jetty where, according to the Jetty coding standard[8] two underscores are used as a prefix for the names of variable static fields and a single underscore for the names of other variable fields.

Some other projects analysed contain names with a range of typographical styles. BCEL, for example, contains a mixture of naming conventions — lower case words separated by underscores and camel case — without a particular scheme prevailing. Without input from the developers it is impossible to understand whether this, apparently inconsistent, mixture of styles is intentional.

B. *Ciphers and Type Acronyms*

The intention of both EJS and JLS is that ciphers are used in formal argument and local variable names. We found ciphers are occasionally used in field names, particularly the declarations of field names of inner classes that provide a generic function, such as string processing, for the outer class, and in classes containing coordinates, such as those in graphics applications or libraries. Rather than being a commonly broken convention, it is a convention that appears to be broken when considered justifiable.

Type acronyms are used extensively in some projects. The distributions shown in Table IV suggest that developers choose to use type acronyms as formal argument and local variable names as suggested by JLS. However, we also found uses of type acronyms in field declarations. Only 6 projects use type acronyms in 2% or more of their field declarations, including ASM and Polyglot where more than 4% of field declarations are type acronyms.

We identified two further unconventional uses of type acronyms. The first is where the acronym relates to a super type, for example the declaration `ServletInputStream is` (Spring). The second is where a type acronym is used as part of a name, for example `StringBuffer filenameSB` (Polyglot).

C. *Redundant Prefixes*

As previously stated, redundant prefixes were found in declarations in some of the projects surveyed. Table VI shows that usage is limited, but that redundant prefixes do play a big role in some projects.

TABLE VI
Distribution of the usage of redundant prefixes

	Field	**Formal Argument**	**Local Variable**
Minimum	.00	.00	.00
1st Quartile	.00	.00	.01
Median	.01	.01	.01
Mean	.05	.01	.01
3rd Quartile	.03	.01	.02
Maximum	.73	.09	.06

JUnit uses redundant prefixes to the greatest extent. 73% of JUnit field declarations — almost all the variable fields declared — are prefixed with `f`. The naming scheme has been revised in a later version than that tested and all redundant prefixes removed. Some 38% of field names declared in Xerces-J are also prefixed with `f`. The convention is applied to most non-constant field declarations, but not consistently[9]. Freemind also uses redundant prefixes in field name declarations (13%) and formal arguments (9.2%), but rarely in local variables (0.8%).

To summarise, we found that a few projects seem to apply inconsistent typography, while there is evidence that others have adopted conventions in addition to those defined in JLS and EJS, including the use of redundant prefixes and alternative typography. We speculate that, in these cases, the JLS and EJS conventions do not always meet developers' needs. We also found that some developers use type acronyms and ciphers in what appear quite sensible ways, but other than intended by JLS. Project specific conventions can be expressed in Nominal. And, as many of the variations of conventions are typographical, names could be refactored automatically as the semantic content of the declarations would not be affected.

VI. Related Work

There is limited literature on the holistic evaluation of the application of naming conventions. Much of the research evaluates the impact of specific aspects of naming conventions, sometimes in combination with other properties of source code, on program comprehension.

[8] https://wiki.eclipse.org/Jetty/Contributor/Coding_Standards

[9] A Google search for *xerces-j coding conventions* reveals that the matter is discussed by Xerces-J contributors

Identifier naming conventions proposed by Relf were evaluated with developers [18]. Relf found that acceptance of specific conventions was related to the developers' experience, suggesting that understanding of the value of conventions increases with experience. The conventions were subsequently applied in a tool to support the creation of names during maintenance and development exercises. The subjects — 69 undergraduate computing students and 10 professional developers — using the tool tended to respond to advice from the tool and created more higher quality names (according to Relf's conventions) than the control group [19].

Two groups have studied the relationship between program comprehension and the use of abbreviations and words in identifier names. Takang, Grubb and Macredie [20] investigated the influence of comments and identifier names containing abbreviations and words on the comprehension of Modula-2 code by undergraduate programmers. Their results showed that a combination of comments and identifier names containing words rather than abbreviations improved program comprehension. However, they also found no statistically significant difference between comprehension of names consisting of abbreviations, and those composed of words.

Lawrie *et al.* [21] revisited this problem area undertaking a similar experiment on the impact of identifier naming on program comprehension. Underlying their approach to their experiment was the understanding that comments are not always present in source code, and that developers have to rely on identifier names alone. Experimental subjects were asked to describe the functionality implemented in one of three variants of methods where identifier names were either single letters, abbreviations or composed of words. The study found that subjects were most confident understanding identifiers containing full words. However, there was, often, only a slight improvement over understanding abbreviated identifiers. Single letter identifiers were the least well understood.

A survey of naming practices in three code bases by Libit *et al.* [7] identified phrasal metaphors used by developers to create names with particular roles. The code surveyed was written in C, C++, C# and Java and consisted of Gnumeric, the Java v1.3 class library, and Microsoft Windows Server 2003. We investigated the use of the metaphors in declarations [13] and applied them in AJC.

CheckStyle [11] and PMD [12] are widely used open source tools that check adherence to coding conventions and include some functionality to evaluate adherence to naming conventions. Both are user configurable, with PMD offering a finer-grained specification of names than CheckStyle. However, both rely on limited property checks — number of characters in a name, for instance — and regular expressions that often do little more than check the capitalisation of the first character of the name, and specify prefixes and suffixes. Neither undertake phrasal analysis, nor distinguish types of content. Both also check for coding style issues such as masking of fields (e.g the declaration of a local variable with the same name as a field in the scope of the field), which we do not assess in this survey. The configuration language in NOMINAL allows conventions to be expressed using a few simple rules, and with much less risk than expressing typography in a regular expression. Other static analysis tools such as FindBugs [22] do offer some limited checks of name quality, but are not easy to configure.

The application of statistical natural language processing techniques by Allamanis *et al.* [23] in NATURALIZE, a coding and naming convention recommendation tool, was motivated, in part, by around 25% of code reviews suggesting changes to naming. Allamanis *et al.* observe that naming conventions, such as JLS and EJS, are prescriptive and project or team conventions tend to evolve by consensus during a project's lifetime and are often not codified. NATURALIZE, however, learns a project's coding and naming styles and recommends refactorings that replicate the conventions used. Consequently it cannot check for adherence to particular conventions and is thus at a disadvantage in the early stages of a project, or where conventions are imposed on a project.

VII. CONCLUSION

Declarations that follow naming conventions are more readable and require less processing by tools, thereby helping support program comprehension, developer onboarding and software maintenance. For Java, references (fields, parameters and variables) constitute around 70% of the name declarations in source code, and have the most diverse and complex typographical and content conventions, but existing tools mostly check a rather small subset of simple typographic conventions.

The contributions made by this paper are:

- a comprehensive survey of the adherence to three sets of naming conventions of reference name declarations in 60 FLOSS Java projects;
- insight into the use of alternative typographical and content conventions by some developers;
- NOMINAL, a configuration language to specify naming conventions and a Java library to evaluate their application;
- NOMINAL definitions for two widely known conventions, EJS and JLS.

The open source nature of NOMINAL[3] and its pre-packaged convention definitions aim to help others adopt, adapt or extend NOMINAL for their own tools and projects.

In our evaluation of 3.5 million reference name declarations we found that the median is over 85% of declarations adhering to conventions, but there is considerable variation in the extent to which developers follow the conventions evaluated, 43–97% for field name typography being the extreme. On inspection, we found occasional projects with inconsistent typography, but many where developers applied a wider range of typographical styles, particularly to differentiate the roles of field declarations, than specified by either JLS or EJS. NOMINAL can be used to specify project specific naming conventions providing opportunities for development teams to codify existing naming conventions for both content and typography and add automated convention checking.

References

[1] 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.

[2] 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.

[3] 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.

[4] E. Enslen, E. Hill, L. Pollock, and K. Vijay-Shanker, "Mining source code to automatically split identifiers for software analysis," in *6th IEEE International Working Conference on Mining Software Repositories*. IEEE, may. 2009, pp. 71–80.

[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] 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.

[8] 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.

[9] ——, "Mining Java class naming conventions," in *Proc. of the 27th IEEE Int'l Conf. on Software Maintenance*. IEEE, 2011, pp. 93–102.

[10] 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.

[11] O. Burn, "Checkstyle," http://checkstyle.sourceforge.net/, 2007.

[12] InfoEther, "PMD," http://pmd.sourceforge.net/, 2008.

[13] S. Butler, M. Wermelinger, and Y. Yu, "A survey of the forms of Java reference names," in *Proc. of the 23rd Int'l Conf. on Program Comprehension*. IEEE, 2015, pp. 196–206.

[14] 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.

[15] D. Klein and C. D. Manning, "Accurate unlexicalised parsing," in *Proc. of the 41st Meeting of the Association for Computational Linguistics*. ACL, 2003, pp. 423–430.

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

[17] 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.

[18] P. A. Relf, "Achieving software quality through identifier names," in *Qualcon 2004*, 2004, presented at Qualcon 2004 http://www.aoq.asn.au/conference2004/conference.html.

[19] ——, "Tool assisted identifier naming for improved software readability: an empirical study," in *Int'l Symp. on Empirical Software Engineering*. IEEE, 2005, pp. 53–62.

[20] A. A. Takang, P. A. Grubb, and R. D. Macredie, "The effects of comments and identifier names on program comprehensibility: an experimental investigation," *J. Prog. Lang.*, vol. 4, no. 3, pp. 143–167, 1996.

[21] D. Lawrie, C. Morrell, H. Feild, and D. Binkley, "What's in a name? A study of identifiers," in *14th IEEE Int'l Conf. on Program Comprehension*. IEEE, 2006, pp. 3–12.

[22] FindBugs, "Find Bugs in Java programs," http://findbugs.sourceforge.net/, 2008.

[23] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, "Learning natural coding conventions," in *Proc. of the 22nd ACM SIGSOFT Int'l Symposium on Foundations of Software Engineering*, ser. FSE 2014. New York, NY, USA: ACM, 2014, pp. 281–293. [Online]. Available: http://doi.acm.org/10.1145/2635868.2635883