Analysing Java Identifier Names in the Wild

Simon Butler

Department of Computing and Communications,
The Open University, Milton Keynes MK7 6AA, United Kingdom

Abstract. Identifier names represent the entities manipulated by a computer program and the actions performed on them and are crucial to program comprehension. This research aims to improve understanding of the forms of identifier name created by software developers and advances techniques for analysing names that can also be applied to improve software engineering tools.

1 Introduction

Source code is the written expression of a software design consisting of identifier names – strings consisting of abbreviations, acronyms and natural language words that represent the entities manipulated by the program and the actions performed upon them – embedded in a formal framework of keywords and operators provided by the programming language. Identifier names are crucial for program comprehension [25], a necessary activity in the development of software. For example names are used to support requirements tracing [2] and concept location for program maintenance [10][17][22], and to determine the consistency of concepts expressed in source code [24][12][23]. The development of techniques to analyse identifier name structure and content [9][21][20][17][7][5] improves understanding of the mechanisms used by developers to encode information in names and increases the accuracy and usefulness of analytical techniques.

Extraction of information from identifier names by software engineering tools is constrained by limitations to the understanding of how developers structure identifier names. Though a lot is known about the structure of method names in Java [20], class and reference (field formal argument and local variable) names — around 62% of unique names and 72% of name declarations in a corpus of 60 FLOSS Java projects [8] — are less well understood and the evidence supporting that understanding is incomplete.

This research attempts to improve understanding of the content and forms of class and reference identifier names created by developers. To understand the variety of names found in multiple Java projects we needed to develop automated tools, which were unavailable when the study started, to extract tokens from names, perform phrasal analysis and to specify and test naming conventions. In Section 2 we describe improvements to existing techniques to split identifier names into their component tokens we made to support our work, and Section 3 outlines techniques we developed to categorise tokens found in names and analyse the phrasal structure of names, where appropriate. Our work has many potential applications for program comprehension tasks, which we illustrate in Section 4 with a system for checking adherence to naming conventions and a proposal for a system for recommending corrections to names. In Section 5 we draw our conclusions and identify some future work.

2 Identifier Name Tokenisation

To access the content of identifier names requires splitting each name into its component tokens. We use the term *token* because names are not always composed of words. For the majority of names (around 80% in our corpus [6][8]) the task of identifying and separating tokens is trivial because developers follow typographical conventions that indicate the boundaries between tokens. In Java, the target programming language of our investigations, two conventions are used. The most common capitalises the initial letter of each token, and is commonly known

as *camel case*, e.g. `isEmpty`, which may be divided into tokens by splitting the name every time a transition from lower to upper case is encountered. The alternative convention, used for constants in Java, is to insert underscores between tokens in upper case, e.g. `BUFFER_SIZE`, which can be split by dividing the name each time an underscore is encountered.

The remaining names contain ambiguous or unclear token boundaries. Some developers, conventionally, capitalise acronyms (e.g. `HTMLEditorKit`), which implies rather than marks a token boundary and, sometimes, do not mark token boundaries (e.g. `commonkeys`). A human can easily divide both names into their tokens, but for software the task is more difficult. A further problem arises from the use of digits in names. Digits may be part of an acronym (e.g. *MP3*, *J2SE* or *3D*), used as a suffix (e.g. `index2`), or a homophone substitution for a preposition (e.g. `html2xml`). Existing techniques for tokenising identifier names developed by Feild *et al.* [13], Enslen *et al.* [11] and Guerrouj *et al.* [15] do not fully address these problems and, at best, treat digits as separate tokens.

We developed an approach to tokenise names containing consecutive upper case characters where the remaining typography is camel case such as `HTMLEditorKit` and `PBinitialize`. The use of two or more consecutive upper case characters indicates that there is a token boundary either before or after the last upper case character in the group. The name is split on both boundaries and the resulting tokens evaluated to determine whether they are recognised words, abbreviations or acronyms, i.e. `HTMLEditorKit` is split to {`html`,`editor`,`kit`} and {`htmle`,`ditor`,`kit`} with the former being chosen as the better solution [6].

Tokenising single case names — by which we mean names without typography that might indicate a token boundary — such as `outputfilename`, employs a similar approach. The name, or part name, must first be tested to determine whether it is a recognised token, or may be a simple neologism created by derivation, e.g. *throwable*, or possibly a spelling mistake. Feild *et al.* created the example of `thenewestone` to illustrate some of the problems to be solved when designing algorithms to tokenise a single case name [13]. We developed algorithms to identify component tokens that improved on existing methods in two ways. Firstly, we examined all feasible tokenisations of a string processing it from left to right and right to left, rather than the existing approach of recursively extracting the longest recognised token while processing the string from left to right [13]. The candidate tokenisations are then evaluated and preference given to tokenisations found by passes in both directions, and then to those containing the fewest whole words. Should a satisfactory tokenisation not be identified, we apply an enhanced version of the algorithm that recursively processes the string by removing initial characters before testing if the remainder of the string begins with a known word. For example a name such as `xxfilenameyy` is tokenised as {`xx`,`file`,`name`,`yy`} following two iterations of the left to right pass where the leading 'x's are removed.

We also developed a set of heuristics for tokenising names containing digits [6]. Initially a test is made for known digit containing acronyms and the name split accordingly. If the digits are not part of a recognised acronym they are separated from the surrounding name, and heuristics used to assign them to the token to the immediate left or right, or retain the digits as a separate token. All the approaches described are implemented in INTT (identifier name tokenisation tool) a Java library used in our research projects[1].

3 Analysis of Name Content and Phrasal Structure

Høst and Østvold undertook extensive analysis of Java method names [19][20] and most of the phrasal structures used by developers are understood. Understanding of Java class and reference names is limited. Singer and Kirkham observed that around 85% of Java class names are composed of one or more nouns preceded, optionally, by one or more adjectives [26]. Liblit *et al.* [21] identified a richer use of phrases in identifier names than predicted by naming conventions, such

[1] INTT is available from `http://oro.open.ac.uk/28352/`

as *The Java Language Specification* [14] and *The Elements of Java Style* [27], particularly for reference and method names, though they did not report detailed quantitative results. Høst and Østvold's and Liblit *et al.*'s work have informed pragmatic approaches to the extraction of information from names by Hill [17] and Abebe and Tonella [1]. While both methods are able to extract information from source code, they rely on constrained models of name structure and are therefore not comprehensive solutions.

Naming conventions state the majority of class and reference names should be nouns or noun phrases, while Liblit *et al.*'s observations give rise to the expectation that some names — boolean references specifically — will be verb phrases such as 'is empty'. Further complications arise for reference names from the content specified in naming conventions: some emphasise use of abbreviations and acronyms derived from type names [14], while others insist on the use of dictionary words as much as possible [27]. We have undertaken studies of Java class [7] and reference names [5] to try to understand their structure in more detail, both in terms of what components are used and, where appropriate, their phrasal structure.

3.1 Class Names

We investigate the lexical and syntactic composition of Java class identifier names in two ways. Firstly, we identify conventional patterns found in the use of parts of speech (PoS). Secondly, we identify the origin of words used in class identifier names within the name of any super class and implemented interfaces to identify patterns of class name construction related to inheritance [7].

Researchers have tried three approaches to PoS tagging identifier names. The first two use off the shelf PoS taggers either alone [17], or in combination with template sentences into which tokenised names are inserted before the sentence is tagged [1]. The third approach is the creation of a PoS tagger specifically for names. Høst and Østvold developed a tagger for Java method identifier names [19], and Gutpa *et al.* created a more generic tagger for names [16].

Available PoS taggers are designed to process sentences, and have been trained on corpora other than identifier names. The Stanford tagger, for example, is trained on a corpus of Wall Street Journal articles. Wall Street Journal articles typically lack the technical vocabulary often found in identifier names, and the sentences in the corpus provide considerably more context than an identifier name for the tagger to use to determine the correct parts of speech.

Our initial experiments with the Stanford tagger found the default model was unreliable and experiments with Abebe and Tonnella's idea of using template sentences [1] to nudge the tagger by providing additional information also seemed not to improve accuracy greatly. We trained a model for the Stanford PoS tagger using 9,000 hand-tagged Java class names. This approach proved more accurate than using the default model with class identifier names, with an accuracy of 87% for class names and 95% for individual tokens.

We analysed 120,000 class names from our corpus of 60 FLOSS Java projects. Analysis of the PoS tags confirmed Singer and Kirkham's observation that 85% of class names are noun phrases. Furthermore we identified another two commonly used patterns of PoS that, together with the noun/noun phrase pattern, account for 90% of the observed class names.

We also undertook a case study of the 652 class names in FreeMind, a mind mapping application, to try to understand the significance of unusually structured class names. We found that 53 names did not match the most common forms of name. While some described actions in the GUI, and others followed project specific naming conventions for small groups of related classes, a few indicated a need for either the name to be refactored to one of the common patterns, or the class to be refactored.

Java classes may extend zero or one super class, and implement zero or more interfaces, or super types. As well as being interested in the phrasal structure of names, we are also interested in the source of name components reused in class names. Developers use a number of patterns including one where the name of the superclass or a supertype is repeated in the name of the

subclass or subtype and preceded by a noun or sometimes adjective denoting the specialisation embodied in the class. The pattern occurs widely, for example in the `java.util` package where the `Map` interface is implemented by the class `HashMap`, among many, which is further extended/specialised by the `java.util.concurrent.ConcurrentHashMap` class. We found a wide variety of forms of reuse of components of superclass/supertype name. Our study only examined a single generation of inheritance and we think that detailed study of naming patterns in inheritance trees will support a more complete understanding of the patterns observed.

3.2 Reference Names

Naming conventions specify a variety of content types for reference names. The Java Language Specification, for example, advocates the use of *ciphers* (well-known single letter abbreviations that represent a generic value of a specific type such as the familiar `i`), *type acronyms* (e.g. `StringBuilder sb`), abbreviations, acronyms, words and phrases [14]. To analyse the phrasal structure of reference names, we first need to identify those names consisting of the content types — acronyms and dictionary words — that might be combined to form phrases. We identified five types of token content specified in naming conventions — cipher, type acronym, abbreviation, acronym and word — a further type, the redundant prefix (a single letter prefix used to indicate type or species), that we have observed in some projects, and a unrecognised category that includes unrecognised abbreviations, neologisms and spelling mistakes. Abbreviations, acronyms and words are identified using MDSC[2], a spell checker developed for identifier names, which incorporates the SCOWL wordlists [3] used by the GNU Aspell spell checker and lists of abbreviations and technical terms from our work and the AMAP project[3] [18]. Categorising tokens allows us to identify four categories of name: two non-phrasal composed of ciphers and type acronyms, one phrasal composed entirely of words and acronyms, and a fourth category where one or more tokens require additional processing, such as abbreviation expansion, checking for spelling errors and neologisms, before the name can be determined to be phrasal.

We analysed 3.5 million reference name declarations and found the majority of projects in our corpus contain a mixture of all four categories of name, with a predominance of phrasal content. The outliers are an important consideration for those building software engineering tools because some projects investigated contain names almost entirely formed of dictionary words and acronyms, while in a few projects 50% or more of name declarations contained at least one unrecognised token.

To analyse the phrasal structure of names we trained another model for the Stanford PoS tagger because the model trained on class names was less accurate with reference names. Again we achieved a similar accuracy for tagging names at 90% and individual tokens at 95%. Using the Stanford parser to identify phrases, we found names consist largely of the phrases observed by Liblit *et al.* However, there are a small proportion of names with different phrasal forms: verb phrases to describe GUI events, prepositional phrases, and complex names consisting of multiple phrases (39 words in one case), most commonly used as identifiers for strings used in GUI messages translated to provide support for internationalisation. We also found a few names consisting of dictionary words with no recognisable phrasal structure, e.g. `isShowLines` [5].

4 Analysis of Adherence to Reference Naming Conventions

Identifier naming conventions are the basis of a uniformity of naming that supports the readability of source code. Naming conventions include typography, content and phrasal structure. Reference names form 52% of the unique names in our corpus and 69% of the declarations.

A significant question arising from the preceding study concerns whether developers use the range of reference name content and phrasal structures according to naming conventions.

[2] MDSC is open source Java software and is available at `https://github.com/sjbutler/mdsc`
[3] `https://msuweb.montclair.edu/~hillem/AMAP.tar.gz`

The preceding study provides a partial answer because we found unanticipated forms of name, particularly the extremely long names sometimes used for strings in resource bundles, and the use of type acronyms as field names.

We developed NOMINAL, a Java library and configuration language[4] that allows users to specify naming conventions and test names for conformance with the conventions. We tested the 3.5 million reference name declarations in our corpus against the naming conventions defined in *The Java Language Specification* [14] and *The Elements of Java Style* [27], and a further set of conventions that reflect the phrase structures observed by Liblit *et al.* [21]. We found that developers do follow conventions, but that they are not always published conventions. Indeed in some projects local conventions on both typography and phrasal content are followed. For example, redundant prefixes are used extensively in a few projects, and typography is sometimes used to highlight details of the declaration, such as the use of the `static` Java modifier [4]. We also found occasional use of type acronyms as field names, which is not specified in naming conventions. Usage appears to be confined to fields of inner classes that provide generic services such as string processing [4].

NOMINAL provides us with an effective tool for the identification names that do not conform to naming conventions related to content and typography. We are working to extend NOMINAL so that the library can recommend corrections to names. The correction of typography is straightforward, where the typographical scheme is well defined. The correction of content, particularly phrasal content, is more challenging because changes to a name's meaning may not reflect the developer's intention.

5 Concluding Remarks and Future Work

Our research has investigated the content and structure of Java class and reference names. To undertake this work we have developed novel approaches to the tokenisation of single case names and names containing digits, and demonstrated the viability of training PoS tagger models for names. We have confirmed and reinforced the findings of Liblit *et al.*, and extended knowledge of the phrasal structures used by developers in class and reference names. We also identified forms of name reuse in class names.

The knowledge acquired is applicable in a number of software engineering scenarios. Improved name splitting and understanding of name content supports better identification of semantic content that might be used for concept or feature location. Feature location may be further improved as a result of a better understanding of the roles words play in a name. Libraries such as NOMINAL and the knowledge of name content and phrase structure can be used to develop tools to support the work of developers and improve the internal quality of software.

There is scope to improve the techniques we developed. Our approach to PoS tagging achieves similar accuracy to approaches tried by other researchers. We believe that the development of a PoS tagger for identifier names could be more accurate, particularly if it is able to recognise type names and treats them as single semantic units. Developers can be creative with language, coining new words to fit their purpose. INTT contains a simple system to identify neologisms created by derivation. However, the development of better techniques for identifying neologisms, especially word blends, could improve the analysis of names.

Throughout this work we have been constantly surprised by the inventive ways in which developers use language in some names, and by the forms of name that they decide are appropriate for their needs. An in depth study of names in multiple projects and programming languages could be used to develop a comprehensive classification of name and token types to support the development of further techniques to process names.

[4] NOMINAL is open source Java software and available from `https://github.com/sjbutler/nominal`

References

1. S.L. Abebe and P. Tonella. Natural language parsing of program element names for concept extraction. In *18th Int'l Conf. on Program Comprehension*, pages 156–159. IEEE, Jun. 2010.
2. G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo. Recovering traceability links between code and documentation. *IEEE Transactions on Software Engineering*, 28(10):970–983, Oct 2002.
3. Kevin Atkinson. SCOWL readme. `http://wordlist.sourceforge.net/scowl-readme`, 2004.
4. Simon Butler, Michel Wermelinger, and Yijun Yu. Investigating naming convention adherence in Java references. Proceedings of the 31st Int'l Conf. on Software Maintenance and Evolution (Forthcoming), 2015.
5. Simon Butler, Michel Wermelinger, and Yijun Yu. A survey of the forms of Java reference names. In *Proc. of the 23rd Int'l Conf. on Program Comprehension*, pages 196–206. IEEE, 2015.
6. Simon Butler, Michel Wermelinger, Yijun Yu, and Helen Sharp. Improving the tokenisation of identifier names. In Mira Mezini, editor, *25th European Conf. on Object-Oriented Programming*, volume 6813 of *Lecture Notes in Computer Science*, pages 130–154. Springer Berlin/Heidelberg, 2011.
7. Simon Butler, Michel Wermelinger, Yijun Yu, and Helen Sharp. Mining Java class naming conventions. In *Proc. of the 27th IEEE Int'l Conf. on Software Maintenance*, pages 93–102. IEEE, 2011.
8. Simon Butler, Michel Wermelinger, Yijun Yu, and Helen Sharp. INVocD: Identifier name vocabulary dataset. In *Proc. of the 10th Working Conf. on Mining Software Repositories*, pages 405–408. IEEE, 2013.
9. Bruno Caprile and Paolo Tonella. Restructuring program identifier names. In *Proc. Int'l Conf. on Software Maintenance*, pages 97–107. IEEE, 2000.
10. Tezcan Dilshener and Michel Wermelinger. Relating developers' concepts and artefact vocabulary in a financial software module. In *Proc. of the 27th IEEE Int'l Conf. on Software Maintenance*, pages 412–417. IEEE, 2011.
11. 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*, pages 71–80. IEEE, may. 2009.
12. J.-R. Falleri, M. Huchard, M. Lafourcade, C. Nebut, V. Prince, and M. Dao. Automatic extraction of a WordNet-like identifier network from software. In *18th Int'l Conf. on Program Comprehension*, pages 4–13. IEEE, jun. 2010.
13. Henry Feild, Dawn Lawrie, and Dave Binkley. An empirical comparison of techniques for extracting concept abbreviations from identifiers. In *Proc. of Int'l Conf. on Software Engineering and Applications*, 2006.
14. James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. *The Java Language Specification (Java SE 8 edition)*. Oracle, Java SE 8 edition, 2014.
15. Latifa Guerrouj, Massimiliano Di Penta, Giuliano Antoniol, and Yann-Gaël Guéhéneuc. TIDIER: an identifier splitting approach using speech recognition techniques. *Journal of Software Maintenance and Evolution: Research and Practice*, 25(6):575–599, 2012.
16. Samir Gupta, Sana Malik, Lori 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*, pages 3–12, 2013.
17. Emily Hill. *Integrating Natural Language and Program Structure Information to Improve Software Search and Exploration*. PhD thesis, The University of Delaware, 2010.
18. Emily Hill, Zachary P. Fry, Haley Boyd, Giriprasad Sridhara, Yana Novikova, Lori 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.*, pages 79–88. ACM, 2008.
19. Einar W. Høst and Bjarte M. Østvold. The Java programmer's phrase book. In *Software Language Engineering*, volume 5452 of *LNCS*, pages 322–341. Springer, 2008.
20. Einar W. Høst and Bjarte M. Østvold. Debugging method names. In *Proc. of the 23rd European Conf. on Object-Oriented Programming*, pages 294–317. Springer-Verlag, 2009.
21. Ben Liblit, Andrew Begel, and Eve 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.
22. A. Marcus and D. Poshyvanyk. The conceptual cohesion of classes. In *Proc, of Int'l Conf. on Software Maintenance*, pages 133–142. IEEE CS, Sept. 2005.
23. Jan Nonnen, Daniel Speicher, and Paul Imhoff. Locating the meaning of terms in source code research on "term introduction". In *Proc. of the 18th Working Conf. on Reverse Engineering*, pages 99–108. IEEE Computer Society, Oct. 2011.
24. Daniel Raţiu. *Intentional Meaning of Programs*. PhD thesis, Technische Universität München, 2009.
25. V. Rajlich and N. Wilde. The role of concepts in program comprehension. In *Proc. 10th Int'l Workshop on Program Comprehension*, pages 271–278. IEEE, 2002.
26. J. Singer and C. Kirkham. Exploiting the correspondence between micro patterns and class names. In *Int'l Working Conf. on Source Code Analysis and Manipulation*, pages 67–76. IEEE, Sept. 2008.
27. Al Vermeulen, Scott W. Ambler, Greg Bumgardner, Eldon Metz, Trevor Misfeldt, Jum Shur, and Patrick Thompson. *The Elements of Java Style*. Cambridge University Press, 2000.