Improving the Tokenisation of Identifier Names

Simon Butler, Michel Wermelinger, Yijun Yu, and Helen Sharp

Computing Department and Centre for Research in Computing
The Open University, Milton Keynes, United Kingdom

Abstract. Identifier names are the main vehicle for semantic information during program comprehension. Identifier names are tokenised into their semantic constituents by tools supporting program comprehension tasks, including concept location and requirements traceability. We present an approach to the automated tokenisation of identifier names that improves on existing techniques in two ways. First, it improves tokenisation accuracy for identifier names of a single case and those containing digits. Second, performance gains over existing techniques are achieved using smaller oracles. Accuracy was evaluated by comparing the output of our algorithm to manual tokenisations of 28,000 identifier names drawn from 60 open source Java projects totalling 16.5 MSLOC. We also undertook a study of the typographical features of identifier names (single case, use of digits, *etc.*) per object-oriented construct (class names, method names, *etc.*), thus providing an insight into naming conventions in industrial-scale object-oriented code. Our tokenisation tool and datasets are publicly available[1].

1 Introduction

Identifier names are strings of characters, often composed of one or more words, abbreviations and acronyms that describe actions and entities in source code. Identifier names are tokenised into their component words to support a wide range of activities in software development, maintenance and research, including concept location [16, 14], to extract semantically useful information for other processes such as traceability [2], and the extraction of domain-specific ontologies [17], or to support investigations of the composition of identifier names [9, 10].

Identifier naming conventions describe how developers *should* construct identifier names. The conventions typically provide mechanisms for identifying boundaries between component words either with separator characters, e.g. `get_text` (Eclipse), or internal capitalisation where the initial letter of the second and successive component words is capitalised, colloquially known as 'camel case', e.g. `getText` (OpenProj). The use of separator characters and internal capitalisation mean identifier names can be readily tokenised. However, a non-negligible proportion of identifier names (we found approximately 15%) are more difficult to tokenise accurately and reliably because they contain features such as upper

[1] http://oro.open.ac.uk/28352/

case acronyms, unconventional uses of capitalisation and digits, or are composed of characters of a single case. Upper case acronyms and words are delimited inconsistently, e.g. `setOSTypes` (jEdit) contains the acronym `OS`, `hasSVUID` (Google Web Toolkit) contains two acronyms, `SVU` and `ID`, concatenated, while `DAYSforMONTH` [7] relies on a change of case to mark a word boundary. Digits are found in some acronyms, e.g. `J2se` and `POP3`, and are also found as discrete tokens, thus there is no simple means of recognising a word boundary where a digit appears in an identifier name. Single case identifier names contain no readily identifiable word boundaries and in some instances, e.g. `ALTORENDSTATE` (JDK), have more than one plausible tokensiation based on dictionary words, which needs to be resolved. Further difficulties arise from the use of mixed case acronyms like OSGi and DnD, where the acronym is difficult to recover as a single token when used in the mixed case form, e.g. as in `isOSGiCompatible` (Eclipse), which lack conventional word boundaries.

Current approaches to identifier name tokenisation [7, 8, 15] report accuracies of around 96% for the tokenisation of unique identifier names. However, some approaches ignore identifier names containing digits [8, 15], or treat digits as discrete tokens [7]. In this paper, we present a step-wise strategy to tokenising identifier names that improves on existing methods [7, 8] in three ways. Firstly, we introduce a method for tokenising single case identifier names that addresses the problem of resolving ambiguous tokenisations and does not rely on the assumption that identifier names begin and end with known words; secondly, we implement and evaluate a method of tokenising identifier names containing digits that relies on an oracle and heuristics; and thirdly, we use an oracle created from published word lists [4] with 117,000 entries, which makes the solution easier to create and deploy than that described in [7] where the oracle consists of 630,000 entries harvested from 9,000 Java projects.

Improvements in identifier name tokenisation can have a big impact on the coverage of concept location and program comprehension tools because tokenisation accuracy is reported in terms of unique identifier names. Hence, even a 1% improvement of accuracy can have a radical effect (e.g. in concept location) if it affects those identifiers with many instances throughout the source code, which would otherwise lead to incorrect or missing concept locations. More importantly, by improving techniques for tokenising identifier names composed of characters of a single case and those containing digits, the coverage of concept location tools can be extended to include identifier names have previously been ignored or underused.

Identifier name tokenisation can also be used in IDE tools to support identifier name quality assurance. For example, some projects use tools like Checkstyle[2] to check conformance to programming conventions when source code is committed to the repository. Such tools typically only ensure typographical conventions, like the usage of word separators in names of constants, not lexical ones, like the usage of dictionary words and recognised abbreviations. Using to-

[2] `http://checkstyle.sourceforge.net/`

kenisation to check whether an identifier name can be properly parsed would allow a more pro-active approach to ensuring the readability of source code.

The remainder of the paper is structured as follows. Section 2 consists of an exposition of the problems encountered when tokenising identifier names. In Section 3 we give an account of related work including the approaches taken by other researchers, before describing our approach to the problem in Section 4. In Section 5 we describe the experiments undertaken to evaluate our solution and compare it with existing solutions. In Sections 6 and 7 we discuss the results of our experiments and draw our conclusions.

2 The Identifier Name Tokenisation Problem

In this section we describe the practical problems encountered when trying to tokenise identifier names.

2.1 The Composition of Identifier Names

Programming languages and programming conventions constrain the content and form of identifier names. Programming languages impose hard constraints, most commonly that identifier names must consist of a single string[3], where the initial character is not a digit, and are composed of a restricted set of characters. For the majority of programming languages, the set of characters permitted in identifier names consists of upper and lower case letters, digits, and some additional characters used as separators. An additional hard constraint imposed by languages such as Perl and PHP is that identifier names begin with specific non-alphanumeric characters used as sigils – signs or symbols – to identify the type represented by the identifier. For example, in Perl '`$`' denotes a scalar and '`@`' a vector.

Programming conventions provide soft constraints in the form of rules on the parts of speech to be used in identifier names, how word boundaries should be constructed and often include the vague injunction that identifier names should be 'meaningful'. Programming conventions typically advise developers to create identifier names with some means of identifying boundaries between words. Java, for example, employs two conventions [19]: constants are composed of words and abbreviations in upper case characters and digits separated by underscores (e.g. `FOO_BAR`), and may be described by the regular expression $U[DU]^*(S[DU]^+)^*$, where D represents a digit, S a separator character and U an upper case letter; and all other identifier names rely on internal capitalisation to separate component words (e.g. `fooBar`).

2.2 Tokenising Identifier Names

Programming conventions, though applied widely, are soft constraints and, consequently, are not applied universally. Thus, tools that tokenise identifier names

[3] Smalltalk method names are a rare exception where the identifier name is separated to accommodate the arguments, e.g. `multiply:` *x* `by:` *y*

need to provide strategies for splitting both conventionally and unconventionally constructed identifier names. Identifier names contain features such as separator characters, changes in case, and digits that have an impact on tokenisation. We discuss each feature before looking at the difficulties encountered when attempting to tokenise identifier names without separator characters or changes in case to indicate word boundaries.

Separator Characters Separator characters – for example, the hyphen in Lisp and the full-stop, or period, in R[4] – can be used to separate the component words in identifier names. Accordingly, the identification of conventional internal boundaries in identifier names is straightforward, and the vocabulary used by the creator of the identifier name can be recovered accurately.

Internal Capitalisation *Internal capitalisation*, often referred to as 'camel case', is an alternative convention for marking word boundaries in identifier names. The start of the second and subsequent words in an identifier name are marked with an upper case letter as in the identifier name `StyledEditorKit` (Java Library), where the boundary between the component words of an identifier name occurs at the transition between a lower case and an upper case letter, i.e. internally capitalised identifier names are of the form $U^?L^+(UL^+)^*$, where L represents a lower case letter, and the word boundary is characterised by the regular expression LU. The word boundary is easily detected and identifier names constructed using internal capitalisation are readily tokenised.

A second type of internal capitalisation boundary is found in practice. Some identifier names contain a sequence consisting of two or more upper case letters followed by at least one lower case letter, i.e. the sequence U^+UL^+. We refer to this type of boundary as the *UCLC boundary*, where UCLC is an abbreviation of upper case to lower case. Most commonly, identifier names with a UCLC boundary contain capitalised acronyms, for example the Java library class name `HTMLEditorKit`. In these cases the word boundary occurs after the penultimate upper case letter of the sequence. However, identifier names have also been found [7] with the same characteristic sequence where the word boundary is marked by the change of case from upper case to lower case, for example `PBinitialize` (Apache Derby). Thus, identification of the UCLC boundary alone is insufficient to support accurate tokenisation [7].

Some identifier names mix the internal capitalisation and separator character conventions, e.g. `ATTRIBUTE_fontSize` (JasperReports). Despite being unconventional, such identifier names pose no further problems for tokenisation than those already given.

Digits Digits occur in identifier names as part of an acronym or as discrete tokens. Where a digit or digits are embedded in the component word, as in the abbreviation `J2SE`, then the boundaries between tokens are defined by the

[4] `http://www.r-project.org/`

internal capitalisation boundaries between the acronym and its neighbours. Abbreviations that have a bounding digit, e.g. `POP3` and `3D`, cannot be separated from other tokens where boundaries are defined by case transitions between alphabetical characters. Even if developers rigorously adopted the convention of only capitalising the initial character of acronyms advocated by Vermeulen [20], that would only help detect the boundary following a trailing digit (e.g. `Pop3Server`), it would not allow the assumption that a leading digit formed a boundary – that is it could not be assumed that UL^+DUL^+ may be tokenised as UL^+ and DUL^+. In other words, because digits do not appear in consistent positions in acronyms, there is no simple rule that can be applied to tokenise identifier names containing acronyms that include digits. Similar complications arise where digits form a discrete component of identifier names, including the use of digits as suffixes (e.g. `index3`) and as homophone substitutions for prepositions (e.g. `html2xml`).

Single Case Some identifier names are composed exclusively of either upper case (U^+) or lower case characters (L^+), or are composed of a single upper case letter followed by lower case letters (UL^+). Such identifier names are often formed from a single word. However, some, such as `maxprefwidth` (Vuze) and `ALTORENDSTATE` (JDK), are composed of more than one word. Lacking word boundary markers, multi-word single case identifier names cannot be tokenised without the application of heuristics or the use of oracles. A variant of the single case pattern is also found within individual tokens in identifier names like `notAValueoutputstream` (Java library), where the developer has created a compound, or failed to mark word boundaries. Accordingly some tokens require inspection and, possibly, further tokenisation. When tokenising identifiers composed of a single case there are two dangers: *ambiguity* and *oversplitting*.

Ambiguity Some single case identifier names have more than one possible tokenisation. For example, `ALTORENDSTATE` is, probably, intended to be interpreted as {`ALT`, `OR`, `END`, `STATE`}. However, it may also be tokenised as {`ALTO`, `RENDS`, `TATE`} by a greedy algorithm that recursively searches for the longest dictionary word match from the beginning of the string, leaving the proper noun 'Tate' as the remaining token. A function of tokenisation tools is therefore to disambiguate multiple tokenisations.

Oversplitting The term *oversplitting* describes the excessive division of tokens by identifier name tokenisation software [7], e.g. tokenising the single case identifier name `outputfilename` as {`out`, `put`, `file`, `name`}. The consequence of this form of oversplitting is that search tools for concept location would not identify that 'output' was a component of `outputfilename` without additional effort to reconstruct words from tokens.

Oversplitting is also practised by developers in two forms: one conventional, the other unconventional. Oversplitting occurs in conventional practice in class

identifier names that are part of an inheritance hierarchy. Class identifier names can be composed of part or all of the super class identifier name that may be consist of a number of tokens and an adjectival phrase indicating the specialisation. For example, the class identifier name `HTMLEditorKit` is composed of part of the type name of its super class `StyledEditorKit` and the adjectival abbreviation `HTML`, yet would be tokenised as {`HTML`, `Editor`, `Kit`}. In this case the compound of the super type is potentially lost, but can be recovered by program comprehension tools. Developers also oversplit components of identifier names unconventionally by inserting additional word boundaries, which increases the difficulty of recovering tokens that reflect the developer's intended meaning. Common instances include the oversplitting of tokens containing digits such as `Http_1_1`, the demarcation of some common prefixes as separate words as in `SubString`, and the division of some compounds such as *metadata* and *uppercase*. In each case, a recognisable semantic unit is subdivided into components and the composite meaning is lost, and must be recovered by program comprehension tools [14].

In the following section we examine the literature on identifier name tokenisation and the approaches adopted by different researchers to solving the problems outlined above.

3 Related Work

Though the tokenisation of identifier names is a relatively common activity undertaken by software engineering researchers [1–3, 6, 9, 11, 14, 16, 18], few researchers evaluate and report their methodologies.

Feild *et al.* [8] conducted an investigation of the tokenisation of single case identifier names, or *hard words* in their terminology. Their experimental effort focused on splitting single case identifier names into component, or *soft*, words. For example, the hard word `hashtable` is constructed from the two soft words `hash` and `table`.

Feild *et al.* compared three approaches to tokenising identifier names – a random algorithm, a greedy algorithm and a neural network. The greedy algorithm applied a recursive algorithm to match substrings of identifier names to words found in the ispell[5] dictionaries to identify potential soft words. For hard words that are composed of more than one soft word, the algorithm starts at the beginning and end of the string looking for the longest known word and repeats the process recursively for the remainder of the string. For example `outputfilename` is tokenised as {`output`, `filename`} from the beginning of the string and as {`outputfile`, `name`} from the end of the string on the first pass. The process is then repeated and the forward and backward components of the algorithm produce the same list of soft words, and thus the single tokenisation {`output`, `file`, `name`}. Where lists of soft words are different, the list containing the higher proportion of known soft words is selected.

[5] `http://www.gnu.org/software/ispell/ispell.html`

Of the three approaches, the greedy algorithm was found to be the more consistent, tokenising identifier names with an accuracy of 75-81%. The greedy algorithm, however, was prone to oversplitting. The neural network was found to be more accurate, but only under particular conditions, for example when the training set of tokenisations was created by an individual.

In a related study Lawrie *et al.* [12] turned to expanding abbreviations to support identifier name tokenisation, and posed the question: how should an ambiguous identifier name such as `thenewestone` be divided into component soft words? Depending on the algorithm used there are a number of plausible tokenisations and no obvious way of selecting the correct one, e.g. {`the`, `newest`, `one`}, {`then`, `ewe`, `stone`}, and {`then`, `ewes`, `tone`}. Lawrie *et al.* suggested that the solution lies in a heuristic that relies on the likelihood of the soft words being found in the vocabulary used in the program's identifier names.

Enslen *et al.* expanded on these ideas in a tool named *Samurai* [7]. Samurai applies a four step algorithm to the tokenisation of identifier names.

1. Identifier names are first tokenised using boundaries marked by separator characters or the transitions between letters and digits.
2. The tokens from step 1 are investigated for the presence of changes from lower case to upper case (the primary internal capitalisation boundary) and split on those boundaries.
3. Tokens found to contain the UCLC boundary – as found in `HTMLEditor` – are investigated using an oracle to determine whether splitting the token following the penultimate upper case letter, or at the change from upper to lower case results in a better tokenisation.
4. Each token is investigated using a recursive algorithm with the support of an oracle to determine whether it can be divided further.

The oracle used in steps 3 and 4 was constructed by recording the frequency of tokens resulting from naive tokenisation based on steps 1 and 2 found in identifier names extracted from 9,000 Sourceforge projects. The oracle returns a score for a token based on its global frequency among all the code analysed and its frequency in the program being analysed. The algorithms in steps 3 and 4 are conservative. In step 3 the algorithm is biased to split the string following the penultimate upper case letter, and will only split on the boundary between upper and lower case where there is overwhelming evidence that the tokenisation is more frequent. The recursive algorithm applied in step 4 will only divide a single case string where there is strong evidence to do so, and also relies on lists of prefixes and suffixes[6] to prevent oversplitting. For example, the token `listen` could be tokenised as {`list`, `en`} for projects where 'list' occurs as a token with much greater frequency than 'listen'. Samurai avoids such oversplitting by ignoring possible tokenisations where one of the candidate tokens, such as 'en', is found in the lists of prefixes and suffixes.

Enslen *et al.* also reproduced the 'greedy algorithm' reported by Feild *et al.* and compared the relative accuracies of the two techniques. The experiment used

[6] Available from `http://www.cis.udel.edu/~enslen/samurai`

a reference set of 8,000 identifier names that had been tokenised by hand. The Samurai algorithm performed better than their implementation of the greedy algorithm, with an accuracy of 97%. The Samurai algorithm has some limitations which we discuss in the next section.

Madani *et al.* [15] developed an algorithm, derived from speech recognition techniques, to split identifier names that does not rely on conventional internal capitalisation boundaries. The approach tries to match substrings of an identifier name with entries in an oracle, both as a straightforward match and through a process of abbreviation expansion analogous to that used by a spell-checking program. Thus `idxcnt` would be tokenised as {`index`, `count`}. Furthermore, because the algorithm ignores internal capitalisation it can consistently tokenise component words such as `MetaData` and `metadata`. Madani *et al.* achieved accuracy rates of between 93% and 96% in their evaluations, which was better than naive camel case splitting in both projects investigated.

In the next section we describe our approach and how it differs from the above techniques.

4 Approach

The approaches described were found to tokenise 96-97% of identifier names accurately. However, there are limitations to each solution and issues with their implementation that make their application in practical tools difficult. Of the three approaches discussed, only Enslen *et al.* attempt to process identifier names containing digits. However, digits are isolated as separate tokens at an early stage of the Samurai algorithm so that meaningful acronyms such as `http11` are tokenised as {`http`, `11`}. Samurai is also hampered by the amount of data collection required to create its supporting oracle.

We have implemented a solution to the problem of identifier name tokenisation that addresses the issues identified in current tools. The solution named INTT, or *I*dentifier *N*ame *T*okeniser *T*ool, is part of a larger source code mining tool [5]. In particular, we have tried to ensure that the solution is relatively easy to implement and deploy, and is able to tokenise all types of identifier name. INTT applies naive tokenisation to identifier names that contain conventional separator character and internal capitalisation word boundaries. Tokens containing the UCLC boundary or digits are processed using heuristics to determine a likely tokenisation, and identifier names composed of letters of a single case are tokenised using an adaptation of the greedy algorithm described above.

The core tokenisation functionality of INTT is implemented in a JAR file so that it can be readily incorporated into other tools. The simple API allows the caller to invoke the tokeniser on a single string, and returns the tokens as an array. Thus front ends can range in sophistication from basic command line utilities that process individual identifier names to parser based tools that process source code. To support programming language independence the set of separator characters can be configured using the API, but the caller is responsible

for removing any sigils from the identifier name. However, INTT has only been tested on identifier names extracted from Java source code.

In summary, our algorithm consists of the following steps, which we discuss in detail below:

1. Identifier names are tokenised using separator characters and the internal capitalisation boundaries.
2. Any token containing the UCLC boundary is tokenised with the support of an oracle.
3. Any identifier names with tokens containing digits are reviewed and tokenised using an oracle and a set of heuristics.
4. Any identifier name composed of a single token is investigated to determine whether it is a recognised word or a neologism constructed from the simple addition of known prefixes and suffixes to a recognised word.
5. Any remaining single token identifier names are tokenised by recursive algorithms. Candidate tokenisations are investigated to reduce oversplitting, before being scored with weight being given to tokens found in the project-specific vocabulary.

4.1 Oracles

To support the tokenisation of identifier names containing the UCLC boundary, digits and single case identifier names, we constructed three oracles: a list of dictionary words, a list of abbreviations and acronyms, and a list of acronyms containing digits. The list of dictionary words consists of some 117,000 words, including inflections and American and Canadian English spelling variations, from the SCOWL package word lists up to size 70, the largest lists consisting of words commonly found in published dictionaries [4]. We added a further 120 common computing and Java terms, e.g. 'arity', 'hostname', 'symlink', and 'throwable'. Previous work [5] included analysis of which identifier names did not correspond to dictionary words and found that several known computing terms were unrecognised. The list of computing terms was hence constructed iteratively over the analysed projects, using the criterion that any word added should be a known, non-trivial computing term. Each oracle was implemented using a Java HashSet so that lookups are performed in constant time.

The use of dictionaries imposes a limitation on the accuracy of the resulting tokenisation because a natural language dictionary cannot be complete. We addressed this limitation by adopting a method to incorporate the lexicon of the program being processed in an additional oracle, which takes a step towards resolving the issue highlighted in Lawrie *et al.*'s question of how to resolve ambiguous tokenisations for identifier names such as `thenewestone` [12]. Tokens resulting from the tokenisation of conventionally constructed identifier names are recorded in a temporary oracle to provide a local – i.e. domain- or project-specific – vocabulary that is employed to support the tokenisation of single case identifier names. For example, tokens extracted from identifier names such as `pageIdx` and `lineCnt` can be used to support the tokenisation of an identifier name like `idxcnt` as {`idx`, `cnt`}.

INTT is also able to incorporate alternative lists of dictionary words in its oracle, and is, thus, potentially language independent. INTT relies on Java's string and character representations, which default to the UTF-16 unicode character encoding standard. So, INTT is able to support dictionaries, and thus tokenise identifier names created using natural languages where all the characters, including accented characters, can be represented using UTF-16 (subject to the constraints on identifier name character sets imposed by the programming language). However, as INTT was designed with the English language and English morphology in mind, adaptation to other languages may not be straightforward.

4.2 Tokenising Conventionally Constructed Identifier Names

The first stage of INTT tokenises identifier names using boundaries marked by separator characters and on the conventional lower case to upper case internal capitalisation boundaries. Where the UCLC boundary is identified, INTT investigates the two possible tokenistations: the conventional internal capitalisation where the boundary lies between the final two letters of the upper case sequence, e.g. as found in `HTMLEditorKit`; and the boundary following the sequence of upper case letters, as in `PBinitialize`. The preferred tokenisation is that containing more words found in the oracle. Where this is not a discriminant, tokenisation at the internal capitalisation boundary is preferred.

Following the initial tokenisation process, identifier names are screened to identify those that require more detailed processing. Identifier names found to contain one or more tokens with digits are tokenised using heuristics and an oracle. Identifier names composed of letters of a single case are tokenised, if necessary, using a variant of the greedy algorithm [12]. These processes are described in detail below.

4.3 Tokenising Identifier Names Containing Digits

In Section 2 we outlined the issues concerning the tokenisation of identifier names containing digits. We identified three uses of digits in identifier names: in acronyms (e.g. `getX500Principal` (JDK)), as suffixes (e.g. `typeList2` (JDK, Java libraries and Xerces)) and as homophone substitutes for prepositions (e.g. `ascii2binary` (JDK and Java libraries)). In the latter two cases the digit, or group of digits, forms a discrete token of the identifier, and if identified correctly the identifier name may be tokenised with relative ease. Acronyms containing digits are more problematic. We have identified two basic forms of acronym: those with an embedded digit, e.g. `J2SE`, and those with one or more bounding digits, e.g. `3D`, `POP3` and `2of7` .

Acronyms with embedded digits are bounded by letters and can be tokenised correctly by relying on internal capitalisation boundaries alone. For example, the method identifier name `createJ2SEPlatform` (Netbeans) can be tokenised as as {`create`, `J2SE`, `Platform`} without any need to investigate the digit. Acronyms with leading or trailing digits cannot easily be tokenised, and neither can those with bounding digits. We made a special case of acronyms with bounding digits.

While they could be tokenised on the assumption that the digits were discrete tokens, we decided that the very few instances of acronyms with bounding digits found in the subject source code were better seen as discrete tokens from a program comprehension perspective. Indeed all the instances we found were noun phrases describing mappings, `1to1`, or bar code encoding schemes `2of7`.

With the exception of the embedded digit form of acronym there is no general rule by which to tokenise identifier names containing digits. Accordingly we created an oracle from a list of common acronyms containing digits and developed a set of heuristics to support the tokenisation of identifier names containing digits.

Identifier names are first tokenised using separator characters and the rules for internal capitalisation. Where a token is found to contain one or more digits it is investigated to determine whether it contains an acronym found in the oracle. Where the acronym is recognised the identifier name is tokenised so that the acronym is a token. For example, `Pop3StoreGBean` can be tokenised using internal capitalisation as {`Pop3Store`, `G`, `Bean`}. The tokens are then investigated for known digit containing acronyms and tokenised on the assumption that `Pop3` is a token, resulting in the tokenisation of {`Pop3`, `Store`}.

Where known acronyms are not found, the digit containing token is split to isolate the digit and an attempt made to determine whether the digit is a suffix of the left hand textual fragment, a prefix of the right hand one, or a discrete token. We employ the following heuristics:

1. If the identifier name consists of a single token with a trailing digit, then the digit is a discrete token, e.g. `radius2` (Netbeans) is tokenised as {`radius`, `2`}.
2. If both the left and right hand tokens are both words or known acronyms the digit is assumed to be a suffix of the left hand token, e.g. `eclipse21Profile` (Eclipse) is tokenised as {`eclipse21`, `Profile`}.
3. If both the left and right hand tokens are unrecognised the digit is assumed to be a suffix of the left hand token, e.g. `c2tnb431r1` (Geronimo and JDK) is tokenised as {`c2`, `tnb431`, `r1`}.
4. If the left hand token is a known word and the right hand token is unrecognised, then the digit is assumed to be a prefix of the right hand token, e.g. `is9x` (Geronimo) is tokenised as {`is`, `9x`}.
5. If the digit is either a 2 or 4 and the left and right hand fragments are known words, the digit is assumed to be a homophone substitution for a preposition, and thus a discrete token, e.g. `ascii2binary` is tokenised as {`ascii`, `2`, `binary`}. It is trivial for the application that calls our tokenisation method to expand the digit into 'to' or 'for', if deemed relevant for the application.

4.4 Tokenising Single Case Identifier Names

To tokenise single case identifier names we adapted the greedy algorithm developed by Feild *et al.* [8]. We identified two areas of the greedy algorithm that required modification to suit our purposes. Firstly, because the algorithm is greedy,

it may fail to identify more accurate tokenisations in particular circumstances. For example, the algorithm finds the longest known word from beginning and end of the string, so `thenewestone` would be tokenised as {`then`, `ewes`, `tone`} by the forward pass, and as {`thenewe`, `stone`} by the backward pass. Secondly, the algorithm assumes that the string to be processed begins or ends with a recognised soft word and therefore cannot locate soft words in a string that both begins and ends with unrecognised words.

Our adaptation of the greedy algorithm is implemented in two forms: greedy and greedier. The greedy algorithm assumes that the string being investigated either begins or ends with a known soft word and the greedier algorithm is only invoked when the greedy algorithm cannot tokenise the string.

Prior to the application of the greedy algorithm, strings are screened to ensure that they are not recognised words or simple neologisms. The check for simple neologisms uses lists of prefixes and suffixes to check that strings are not composed of a combination of, for example, a known prefix followed by a known word. This allows identifier names such as `discontiguous` (Java Libraries, JDK and NetBeans) to be recognised as words, despite them not being recorded in the dictionary. The greedy algorithm iterates over the characters of the identifier name string forwards (see Algorithm 1) and backwards. On each iteration, the substring from the end of the string to the current character is tested using the dictionary words and acronyms oracles to establish whether the substring is a known word or acronym. When a match is found the soft word is stored in a list of candidates and the search invoked recursively on the remainder of the string. Where no word can be identified the remainder of the string is added to the list of candidates.

Algorithm 1 INTT greedy algorithm: forward tokenisation pass

```
 1: procedure greedyTokeniseForwards(s)
 2:     candidates                                          ▷ a list of lists
 3:     for i ← 0, length(s) do
 4:         if s[0, i] is found in dictionary then
 5:             rightCandidates ← greedyTokeniseForwards(s[i + 1, length(s)])
 6:             for all lists of tokens in rightCandidates do
 7:                 add s[0, i] to beginning of list
 8:                 add list to candidates
 9:             end for
10:         end if
11:     end for
12:     if candidates is empty then
13:         create new list with s as member
14:         add list to candidates
15:     end if
16:     return candidates
17: end procedure
```

When the greedy algorithm is unable to tokenise the string, the greedier algorithm is invoked. The greedier algorithm attempts to tokenise a string by creating a prefix of increasing length from the initial characters and invokes the greedy algorithm on the remainder of the string to identify known words (see Algorithm 2). For example, for the string `cdoutputef`, `c` is added to a list of candidates and the greedy algorithm invoked on `doutputef`, then the prefix `cd` is tried and the greedy algorithm invoked on `outputef` resulting in the tokenisation {`cd`, `output`, `ef`}. This process is repeated, processing the string both forwards and backwards until the prefix and suffix are one character less than half the length of the string being tokenised, which allows the forward and backward passes to find small words sandwiched between long prefixes and suffixes, while avoiding redundant processing. For example in the string `yyytozzz` both the forwards and backwards passes will recognise `to`, and in the string `yyyytozz` the backwards pass will recognise `to`.

Algorithm 2 INTT greedier algorithm: backwards tokenisation pass

```
 1: procedure greedierTokeniseBackwards(s)
 2:     candidates                                          ▷ a list of lists
 3:     for i ← length(s), length(s)/2 do
 4:         leftCandidates ← greedyTokeniseBackwards(s[0, i − 1])
 5:         for all lists of tokens in leftCandidates do
 6:             add s[i, length(s)] to beginning of list
 7:             add list to candidates
 8:         end for
 9:     end for
10:     return candidates
11: end procedure
```

Each list of candidate component words is scored according to the percentage of the component words found in the dictionaries of words and abbreviations, and the program vocabulary – i.e. component words found in identifier names in the program that were split using conventional internal capitalisation boundaries and separator characters. The percentage of known words is recorded as an integer and a weight of one added for each word found in the program vocabulary. For example, suppose splitting `thenewestone` resulted in two candidate sets {`the, newest, one`} and {`then, ewe, stone`}. All the words in both sets are found in the dictionaries used and thus each set of candidates score 100. However, suppose `newest` and `one` are found in the list of identifier names used in the program, so two is added to the score of the first set, and that is selected as the preferred tokenisation.

The algorithm, because of its intensive search for candidate component words, is prone to evaluating an oversplit tokenisation as a better option than a more plausible tokenisation. To reduce oversplitting, each candidate tokenisation is examined prior to scoring to determine whether adjacent soft words can be con-

catenated to form dictionary words. Where this is the case the oversplit set of tokens is replaced by the concatenated version. For example `outputfile` would be tokenised as {`output`, `file`} and {`out`, `put`, `file`}. Following the check for oversplitting, the first two tokens of the latter tokenisation would be concatenated making the two tokenisations identical, allowing one to be discarded.

The key advantage offered by the greedy and greedier algorithms are that a single case identifier name can be tokenised without the requirement that it begins or ends with a known word. For example, Feild *et al.*'s greedy algorithm cannot tokenise identifier names like `lboundsb` unless 'b' or 'l' are separate entries in the oracle. Samurai can only tokenise `lboundsb` if 'l' or 'lbounds' are found as separate tokens in the oracle. Our algorithm can tokenise `lboundsb` using a dictionary where 'bounds' is an entry.

In the following section we evaluate the accuracy of our identifier name tokenisation algorithm and compare its performance with Samurai and Feild *et al.*'s greedy algorithm.

5 Experiments and Results

To evaluate our approach and compare its performance with existing tools we adopted a similar procedure to that used by Feild *et al.* [8] and Enslen *et al.* [7]. However, instead of using a single test set of identifier names, we created seven test sets consisting of 4,000 identifier names each, extracted at random from a database of 827,475 unique identifier names from 16.5 MSLOC[7] of Java from 60 projects, including ArgoUML, Cobertura, Eclipse, FindBugs, the Java libraries and JDK, Kawa and Xerces[8]. One test set consists of identifier names selected at random from the database. Five test sets consist of random selections of particular *species* of identifier name – we use the term *species* to identify the role the identifier name plays in the programming language, such as a class or method name. The seventh set consists of identifier names composed of a single case only (see Table 1).

Each test set of 4,000 identifier names was tokenised manually by the first author to provide reference sets of tokenisations. The resulting text files consist of lines composed of the identifier name followed by a tab character and the tokenised form of the identifier name, normalised in lower case, with each token separated by a dash, e.g. `HTMLEditorKit`⟨*tab*⟩`html-editor-kit`. Bias may have been introduced to our experiment by the reference tokenisations having not been created independently and we discuss the implications below in Subsection 5.4 Threats to Validity.

The identifier names in the test sets were classified using four largely mutually exclusive categories that reflect particular features of identifier name composition related to the difficulty of accurate tokenisation. The categories are:

[7] Obtained using Sloccount `http://www.dwheeler.com/sloccount/`

[8] A complete list of the projects analysed is available with the INTT library at `http://oro.open.ac.uk/28352/`

– **Conventional** identifier names are composed of groups of letters divided by internal capitalisation (lower case to upper case boundary) or separator characters.
– **Digits** identifier names contain one or more digits.
– **Single case** identifier names are composed only of letters of the same case, or begin with a single upper case letter with the remaining characters all lower case.
– **UCLC** identifier names contain two or more contiguous upper case characters followed by a lower case character.

Identifiers names are categorised by first testing for the presence of one or more digits, then testing for the UCLC boundary. Consequently the digits category may contain some identifier names that also have the UCLC boundary. In the seven test sets there are a total of 1768 identifier names containing digits, of which 62 also contain a UCLC boundary. The classification system is intended to allow the exclusion of identifier names containing digits from evaluations of those tools that do not attempt realistic tokenisation of such identifier names, and to allow evaluation of our approach to tokenising identifier names containing digits. The distribution of the four categories of identifier names in each of the datasets is given in Table 1.

Table 1. Distribution of identifier name categories in datasets

Dataset	Description	Conventional	Digits	Single Case	UCLC
A	Random identifier names	2414	467	1011	106
B	Class names	3133	185	113	569
C	Method names	3459	116	184	151
D	Field names	2717	401	818	64
E	Formal arguments	2754	250	961	34
F	Local variable names	2596	349	1021	34
G	Single case	0	0	4000	0

We also surveyed the 60 projects in our database. Figure 1 shows the distribution of each category as a proportion of the total number of unique identifier names in each application. Identifier names containing only conventional boundaries are by far the most common form of identifier name found in all the projects surveyed. A significant proportion of single case identifier names are found in most projects, and around 10% of identifier names contain digits or the UCLC boundary. Table 2 gives a breakdown of the proportion of unique identifier names

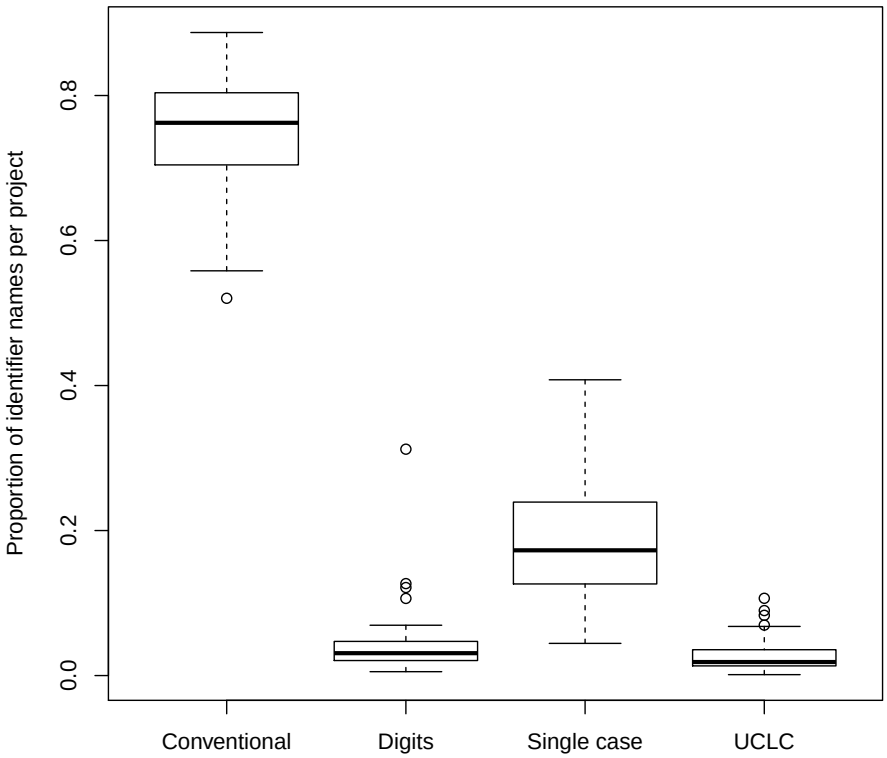

Fig. 1. Distribution of the percentage of unique identifier names found in each category for sixty Java projects

in each category across all 60 projects for each species of identifier. Test sets B to F reflect the most common species, with the exception of constructor names which are lexically identical to class identifier names, but differ in distribution because not all classes have an explicitly declared constructor, while others have more than one.

Table 2 shows that identifier names containing digits and those containing UCLC boundaries constitute nearly 9% of all the identifier names surveyed. Class, constructor and interface identifier names, the most important names for high level global program comprehension, have a relatively high incidence of identifier names containing the UCLC boundary – 13% for class and constructor identifier names and 32% for interface identifier names. In other words, approximately 20% of class names and 40% of interface names require more sophisticated heuristics to determine how to tokenise them.

We evaluated the performance of INTT by assessing the accuracy with which the test sets of identifier names were tokenised, and by comparing INTT with an implementation of the Samurai algorithm, both in terms of accuracy and the relative strengths and weaknesses of the two approaches.

Table 2. Percentage distribution of identifier name categories by species

Species	Conventional	Digits	Single case	UCLC	Overall %
Annotation	70.4	0.2	25.6	3.8	**0.1**
Annotation member	49.8	0.5	49.5	0.2	**<0.1**
Class	79.8	4.1	2.9	13.2	**9.8**
Constructor	79.8	3.5	3.1	13.5	**7.2**
Enum	73.4	0.5	19.4	6.7	**0.1**
Enum constant	55.9	10.2	33.6	0.2	**0.8**
Field	86.1	6.0	6.2	1.7	**27.1**
Formal argument	81.8	3.0	14.2	0.1	**8.1**
Interface	59.3	2.6	6.4	31.7	**1.5**
Label name	59.1	15.7	25.0	0.1	**0.1**
Local variable	82.4	3.8	12.6	1.2	**16.9**
Method	91.6	2.9	1.6	3.9	**28.4**
Total	**84.9**	**4.1**	**6.4**	**4.6**	

5.1 INTT

We used INTT to tokenise the identifier names in each of the seven datasets. The accuracy of the tokenisations was automatically checked against the reference tokenisations for each dataset using a small Java program. A percentage accuracy score calculated for INTT's overall performance and for each species of identifier name. A percentage accuracy was also calculated for each of the four structural categories found in each set of identifier names, see Table 3. (The results for dataset G are reported in Subsection 5.3.)

INTT was found to have an overall accuracy of 96-97%, which improves marginally when identifier names containing digits are excluded. Identifier names containing digits are tokenised with an accuracy in excess of 85% for three of the six data sets A–F. However, accuracy drops to 64% for method identifier names containing digits. Inspection of the tokenisations for class and method names show that there are two contributing factors: firstly, the assumption that a recognised acronym containing digits always takes precedence over the heuristics when determining a tokenisation led to incorrect tokenisations in some instances and, secondly, some oversplitting of textual tokens occurs. An example of the former is the method name `replaceXpp3DOM` (NetBeans) which was tokenised as {`replace`, `Xpp`, `3D`, `OM`} on the basis that `3D` is a known acronym containing digits. Applying the heuristics alone, however, would have found the correct tokenisation of {`replace`, `Xpp3`, `DOM`}.

Table 3. Percentage accuracies for INTT

Dataset		Conventional	Digits	Single case	UCLC	Overall	Without digits
A	Random identifier names	97.3	95.9	97.4	85.8	**96.9**	**97.0**
B	Class names	98.3	85.4	92.4	92.1	**96.5**	**97.1**
C	Method names	97.1	63.8	96.8	92.7	**96.0**	**96.9**
D	Field names	97.5	88.7	96.4	87.5	**96.3**	**97.1**
E	Formal arguments	98.8	94.4	93.4	79.4	**97.0**	**97.2**
F	Local variable names	98.2	94.3	92.0	85.3	**96.2**	**96.3**

The overall percentage accuracy for each dataset is comparable with the accuracies reported for the Samurai tool [7] (97%) and by Madani *et al.* [15] (93-96%). The breakdowns for each structural type of identifier name show that INTT performs less consistently for identifier names containing digits and for those containing the UCLC boundary.

5.2 Comparison with Samurai

To make a comparison with the work of Enslen *et al.* we developed an implementation of the Samurai tool based on the published pseudocode and textual descriptions of the algorithm [7]. The implementation processed the seven test sets of identifier names and the resulting tokenisations were scored for accuracy against the reference tokenisations. The results are shown in Table 4 with the exception of the single case dataset G, which is reported below in Subsection 5.3. The overall accuracy figure given for our implementation of the Samurai algorithm in Table 4 excludes identifier names with digits, and should be compared with the figures in the rightmost column of Table 3. Samurai's treatment of digits as discrete tokens leads to an accuracy of 80% or more for all but class and method identifier names, where accuracy falls to 45% and 55% respectively.

Our implementation of the Samurai algorithm performs less well than the original [7]. On inspecting the tokenisations we found more oversplitting than we had anticipated. There are a number of factors that could contribute to the observed difference in performance, which we discuss in Subsection 5.4 Threats to Validity.

5.3 Single case identifier names

Both INTT and Samurai contain algorithms for tokenising single case identifier names that are intended to improve on Feild *et al.*'s greedy algorithm. To

Table 4. Percentage accuracies for Samurai

Dataset		Conventional	Digits	Single case	UCLC	Without digits
A	Random identifier names	93.3	92.9	69.1	82.1	**86.3**
B	Class names	94.0	44.9	86.3	81.5	**91.7**
C	Method names	92.8	55.2	88.8	83.4	**92.3**
D	Field names	91.3	78.8	78.2	73.4	**87.7**
E	Formal arguments	94.8	88.4	75.0	64.7	**89.4**
F	Local variable names	92.7	86.2	67.7	70.6	**85.4**

compare the two tools we extracted a data set of 4,000 random single case identifier names from our database. All the identifier names consist of a minimum of eight characters: 2,497 are composed of more than one word or abbreviation, the remainder are either single words found in the dictionary or have no obvious tokenisation.

We implemented the greedy algorithm developed by Feild *et al.* following their published description [8], to provide a baseline of performance from which we could evaluate the improvement in performance represented by INTT and Samurai. The supporting dictionary for the Feild *et al.*'s greedy algorithm was constructed from the English word lists provided with ispell v3.1.20, the same version used by Feild *et al.*. We replaced their stop-list and list of abbreviations, with the same list of abbreviations used in INTT and the additional list of terms that are included in INTT's dictionary.

Enslen *et al.* found that Samurai and greedy both had their strengths. Samurai is a conservative algorithm that tokenises identifier names only when the tokenisation is a very much better option than not tokenising. As a result, the greedy algorithm correctly tokenised identifier names that Samurai left intact. However, the greedy algorithm was more prone to oversplitting than the more conservative Samurai [7].

The 4,000 single case identifier names were tokenised with 78.4% accuracy by our implementation of the 'greedy' algorithm, with 70.4% accuracy by our implementation of Samurai, and with 81.6% accuracy by INTT.

5.4 Threats to Validity

The threats to validity in this study are concerned with construct validity and external validity. We do not consider internal validity because we make no claims

of causality. Similarly, we do not consider statistical conclusion validity, because we have not used any statistical tests.

Construct Validity There are two key concerns regarding construct validity: the possibility of bias being introduced through manual tokenisation of identifier names used to create sets of reference tokenisations; and the observed difference in perfomance between our implementation of Samurai and the accuracy reported for the original implementation [7].

That we split the identifier names for the reference tokenisations ourselves may have introduced a bias towards tokenisations that favour our tool. We guarded against this during the manual tokenisation process as much as possible, and conducted a review of the reference sets to look for any possible bias and revised any such tokenisations found. Of the related works [8, 7, 15] only Enslen *et al.* used a reference set of tokenisations created independently.

We have identified three factors that may explain the reduced accuracy achieved by our implementation of Samurai in comparison to the reported accuracy of the original. When implementing the Samurai algorithm, we took all reasonable steps, including extensive unit testing, to ensure our implementation conformed to the published pseudo code and text descriptions [7]. However, it is possible that we may have inadvertently introduced errors. There is the possibility that computational steps may have inadvertently been omitted from the published pseudo code description. The third possibility is that the scoring formula used in Samurai to identify preferable tokenisations, which was derived empirically, may not hold for oracles composed of fewer tokens with lower frequencies. The oracle used in our implementation of Samurai was constructed using identifier names found in 60 Java projects, much fewer than the 9,000 projects Enslen *et al.* used as the basis for their dictionary. Our version of the Samurai oracle contains 61,580 tokens, with a total frequency of 3 million. In comparison the original Samurai oracle was created using 630,000 tokens with a total frequency of 938 million.

External Validity External validity is concerned with generalisations that may be drawn from the results. Our experiments were conducted using identifier names extracted from Java source code only. Although we cannot claim any accuracy values for other programming languages, we would expect results to be similar for programming languages with similar programming conventions, because our tokenisation approach is independent of the programming language. Our experiments were also conducted on identifier names constructed using the English language. While the techniques and the tool we developed can be applied readily to identifier names in other natural languages, some of the heuristics, in particular the treatment of '2' and '4' as homophone substitutions for prepositions, may need to be revised for non-English natural languages.

6 Discussion

One of our primary motivations for adopting the approach described above was a concern over the computing resources, both in terms of time and space that were being devoted to solving the problem of identifier name tokenisation. The approach taken by Madani *et al.* processes each identifier name in detail and is thus relatively computationally intensive, while the Samurai algorithm relies on harvesting identifier names from a large body of existing source code – a total of 9,000 projects – to create the supporting oracle. Like Samurai, we process identifier names selectively and reserve more detailed processing for those identifier names assumed to be more problematic. However, we achieve levels of accuracy similar to the published figures for Samurai using a smaller oracle constructed, largely, from readily available components such as the SCOWL word lists.

6.1 Identifier names containing digits

We demonstrated an approach to tokenising identifier names containing digits that achieves an accuracy of 64% at worst and most commonly 85%-95%. The only tool available for comparison was our implementation of the Samurai algorithm, which takes a simple and unambiguous approach to tokenising identifier names containing digits and achieves, an accuracy that is consistently between 10% and 3% less than that achieved by INTT, with the exception of class identifier names where Samurai's treatment of digits as discrete tokens results in an accuracy of 45%, some 40% less than INTT.

While we are largely satisfied with having achieved such high rates of accuracy, there is room for improvement. Inspection of INTT's output showed that some inaccurate tokenisations could be attributed to incorrect tokenisation of textual portions of the identifier name. However, they also showed that some of our heuristics for identifying how to tokenise around digits require refinement. One possibility is the introduction of a specific heuristic for tokens of the form 'v5', signifying a version number, so that they are tokenised consistently. We found that though most were tokenised accurately, some identifier names, for example `SPARCV9FlushwInstruction` (JDK), were not. The difficulty appears not to be the digit alone, but that the digit in combination with the letter is key to accurate tokenisation. Other incorrect tokenisations occurred where identifier names such as `replaceXpp3DOM` contain a known acronym. The solution in such cases appears to be to choose between the tokenisation resulting from using recognised acronyms, and that arising from the application of the heuristics alone.

6.2 Limitations

No current approach tokenises all identifier names accurately. Indeed, accurate tokenisation of all identifier names may only be possible with some projects where a given set of identifier naming conventions are strictly followed. However, we

would argue that there are a number of barriers to tokenisation that are difficult to overcome, and outside the control of those processing source code to extract information. An underlying assumption of the approaches taken to identifier name tokenisation is that identifier names contain semantic information in the form of words, abbreviations and acronyms and that these can be identified and recovered. Developers, however, do not always follow identifier naming conventions and building software that can process all the forms of identifier names that developers can dream up is most likely impossible and would require a great deal of additional effort for a minimal increase in accuracy. For example, `is0x8000000000000000L` (Xerces) is an extremely unusual form of identifier name – the form is seen only three times[9] in the 60 projects we surveyed – which would require additional functionality to parse the hexadecimal number in order to tokenise the identifier name accurately.

Another limitation arises from neologisms and misspelt words. Neologisms found in the single case test set include 'devoidify', 'detokenated', 'discontiguous', 'grandcestor', 'indentator', 'pathinate' and 'precisify'. With the exception of 'grandcestor' these are all formed by the unconventional use of prefixes and suffixes with recognised words or morphological stems. Some, e.g. 'discontiguous' are vulnerable to oversplitting by the greedy algorithm, and algorithms based on it. Others may cause problems when concatenated with other words in single case identifier names where a plausible tokenisation is found to span the intended boundary between words.

Samurai and INTT both guard against oversplitting neologisms by using lists of prefixes and suffixes. INTT identifies single case identifier names found to be formed by a recognised word in combination with either or both a known prefix or suffix and does not attempt to tokenise them. Samurai tries to tokenise all single case identifier names, but rejects possible tokenisations where one of the resulting tokens would be a known prefix or suffix. All of the neologisms listed would be recognised as single words by both approaches. However, INTT would not recognise 'precisify' as a neologism resulting from concatenation and would try to tokenise it.

Tools that use natural language dictionaries as oracles will try to tokenise a misspelt word, whether it is found in isolation or concatenated with another word, as a single case identifier name. The majority of observed misspellings result from insertion of an additional letter, omission of a letter or transposition of two letters. Precisely the sort of problem that can be readily identified by a spell checker. For example, `possition` (NetBeans) is oversplit by both INTT and the greedy algorithm as {`pos`, `sit`, `ion`} and {`poss`, `it`, `ion`}, respectively. Samurai also oversplits `possition` probably because of a combination of the relative rarity of the spelling mistake, the more common occurrence of the token `poss` (AspectJ, Eclipse, Netbeans, and Xalan). A step towards preventing some oversplitting of misspelt words could be achieved through the use of algorithms applied in spell-checking software, such as the Levenshtein distance [13].

[9] NetBeans unit tests include the method names `test0x01` and `test0x16`.

Inspection of the tokenisations of the test sets for each tool show that the greedy algorithm is prone to oversplitting neologisms particularly where a suffix such as 'able' that is also a word has been added to a dictionary word, e.g. `zoomable` (JFreeChart). Greedy also cannot consistently tokenise identifier names that start and end with abbreviations not found in its dictionary, e.g. `tstampff` (BORG Calendar), and cannot differentiate between ambiguous tokenisations. Indeed, Feild *et al.* provide no description of how to differentiate between tokenisations that return identical scores [8]. In our implementation of the greedy algorithm, the tokenisation resulting from the backward pass is selected in such situations, because English language inflections, particularly the single 's', can be included by the forward pass of the algorithm. For example, `debugstackmap` (JDK) is tokenised incorrectly as {`debugs`, `tack`, `map`} by the forward pass and correctly as {`debug`, `stack`, `map`} by the backward pass. The backward pass is also prone to incorrect tokenisations, though from inspection of the test set this is much less common. For example, the reverse pass tokenises `commonkeys` (JDK) as {`com`, `monkeys`}, using ispell word lists where 'com' is listed as a word.

Tools such as INTT and Samurai work on the assumption that developers generally follow identifier naming conventions and that computational effort is required for exceptions that can be identified. As noted in our description of the problem (see Section 2) the assumption is an approximation. There are many cases where the conventions on word division are broken, or are used in ways that divide the elements of semantic units so as to render them meaningless. In other words, a key issue for tokenisation tools is that word divisions, be they separator characters or internal capitalisation, can be misleading and are thus not always reliable. Consequently, meaningful tokens may need to be reconstructed by concatenating adjacent tokens.

7 Conclusions

Identifier names are the main vehicle for semantic information during program comprehension. The majority of identifier names consist of two or more words or acronyms concatenated and therefore need to be tokenised to recover their semantic constituents, which can then be used for tool-supported program comprehension tasks, including concept location and requirements traceability. Tool-supported program comprehension is important for the maintenance of large object-oriented software projects where cross-cutting concerns mean that concepts are often not located in a single class, but are found diffused through the source code.

While identifier naming conventions should make the tokenisation of identifier names a straightforward task, they are not always clear, particularly with regard to digits, and developers do not always follow conventions rigorously, either using potentially ambiguous word division markers or none at all. Thus accurate identifier name tokenisation is a challenging task.

In particular, the tokenisation of identifier names of a single case is non-trivial and there are known limitations to existing methods, while identifier names containing digits have been largely ignored by published methods of identifier name tokenisation. However, these two forms of identifier name occur with a frequency of 9% in our survey of identifier names extracted from 16.5 MSLOC of Java source code, demonstrating the need to improve methods of tokenisation.

In this paper we make two contributions that improve on current identifier name tokenisation practice. First, we have introduced an original method for tokenising identifier names containing digits that can achieve accuracies in excess of 90% and is a consistent improvement over a naive tokenisation scheme. Second, we demonstrate an improvement on current methods for tokenising single case identifier names, on the one hand in terms of improved accuracy and scope by tokenising forms of identifier name that current tools cannot, and on the other hand in terms of resource usage by achieving similar or better accuracy using an oracle with less than 20% of the entries. Furthermore, the oracle we used can be constructed easily from available components, whereas the Samurai algorithm relies on identifier names harvested from 9,000 Java projects.

We make two further contributions. Firstly, INTT, written in Java, is available for download[10] as a JAR file with an API that allows the identifier name tokenisation functionality described in this paper to be integrated into other tools. Secondly, the data used in this study is made available as plain text files. The data consists of the seven test datasets of 28,000 identifier names together with the manually obtained reference tokenisations, and 1.4 million records of over 800,000 unique identifier names in 60 open source Java projects, including information on the identifier species. By making these computational and data resources available, we hope to contribute to the further development of identifier name based techniques (not just tokenisation) that help improve software maintenance tasks.

Acknowledgements We would like to thank the anonymous reviewers on the ECOOP 2011 Program Committee, and Tiago Alves and Eric Bouwers for their thoughtful comments that have helped improve this paper.

References

1. Abebe, S., Tonella, P.: Natural language parsing of program element names for concept extraction. In: 18th Int'l Conf. on Program Comprehension. pp. 156–159. IEEE (jun 2010)
2. Antoniol, G., Canfora, G., Casazza, G., De Lucia, A., Merlo, E.: Recovering traceability links between code and documentation. IEEE Transactions on Software Engineering 28(10), 970–983 (Oct 2002)
3. Antoniol, G., Gueheneuc, Y.G., Merlo, E., Tonella, P.: Mining the lexicon used by programmers during sofware [sic] evolution. In: Proc. of Int'l Conf. on Software Maintenance. pp. 14–23. IEEE (Oct 2007)

[10] `http://oro.open.ac.uk/28352/`

4. Atkinson, K.: SCOWL readme. `http://wordlist.sourceforge.net/scowl-readme` (2004)
5. Butler, S., Wermelinger, M., Yu, Y., Sharp, H.: Exploring the influence of identifier names on code quality: an empirical study. In: Proc. of the 14th European Conf. on Software Maintenance and Reengineering. pp. 159–168. IEEE Computer Society (2010)
6. Caprile, B., Tonella, P.: Nomen est omen: analyzing the language of function identifiers. In: Proc. Sixth Working Conf. on Reverse Engineering. pp. 112–122. IEEE (Oct 1999)
7. Enslen, E., Hill, E., Pollock, L., Vijay-Shanker, K.: Mining source code to automatically split identifiers for software analysis. In: 6th IEEE International Working Conference on Mining Software Repositories. pp. 71 –80. IEEE (may 2009)
8. Feild, H., Lawrie, D., Binkley, D.: An empirical comparison of techniques for extracting concept abbreviations from identifiers. In: Proc. of Int'l Conf. on Software Engineering and Applications (2006)
9. Høst, E.W., Østvold, B.M.: The Java programmer's phrase book. In: Software Language Engineering. LNCS, vol. 5452, pp. 322–341. Springer (2008)
10. Høst, E.W., Østvold, B.M.: Debugging method names. In: Proc. of the 23rd European Conf. on Object-Oriented Programming. pp. 294–317. Springer-Verlag (2009)
11. Kuhn, A., Ducasse, S., Gírba, T.: Semantic clustering: Identifying topics in source code. Information and Software Technology 49(3), 230–243 (2007)
12. Lawrie, D., Feild, H., Binkley, D.: Quantifying identifier quality: an analysis of trends. Empirical Software Engineering 12(4), 359–388 (2007)
13. Levenshtein, V.I.: Binary codes capable of correcting deletions, insertions, and reversals. Cybernetics and Control Theory 10(8), 707–710 (1966)
14. Ma, H., Amor, R., Tempero, E.: Indexing the Java API using source code. In: 19th Australian Conf. on Software Engineering. pp. 451–460 (March 2008)
15. Madani, N., Guerrouj, L., Penta, M.D., Guéhéneuc, Y.G., Antoniol, G.: Recognizing words from source code identifiers using speech recognition techniques. In: Proc. of the Conf. on Software Maintenance and Reengineering. pp. 69–78. IEEE (2010)
16. Marcus, A., Rajlich, V., Buchta, J., Petrenko, M., Sergeyev, A.: Static techniques for concept location in object-oriented code. In: Proc. 13th Int'l Workshop on Program Comprehension. pp. 33–42. IEEE (May 2005)
17. Raţiu, D., Feilkas, M., Jürjens, J.: Extracting domain ontologies from domain specific apis. In: Proc. of the 12th European Conf. on Software Maintenance and Reengineering. pp. 203–212. IEEE Computer Society (2008)
18. Singer, J., Kirkham, C.: Exploiting the correspondence between micro patterns and class names. In: Int'l Working Conf. on Source Code Analysis and Manipulation. pp. 67–76. IEEE (Sept 2008)
19. Sun Microsystems: Code conventions for the Java programming language. `http://java.sun.com/docs/codeconv` (1999)
20. Vermeulen, A., Ambler, S.W., Bumgardner, G., Metz, E., Misfeldt, T., Shur, J., Thompson, P.: The Elements of Java Style. Cambridge University Press (2000)