

# Mining Source Code to Automatically Split Identifiers for Software Analysis \*

Eric Enslin, Emily Hill, Lori Pollock and K. Vijay-Shanker  
Department of Computer and Information Sciences  
University of Delaware  
Newark, DE 19716 USA  
{enslen, hill, pollock, vijay}@cis.udel.edu

## Abstract

*Automated software engineering tools (e.g., program search, concern location, code reuse, quality assessment, etc.) increasingly rely on natural language information from comments and identifiers in code. The first step in analyzing words from identifiers requires splitting identifiers into their constituent words. Unlike natural languages, where space and punctuation are used to delineate words, identifiers cannot contain spaces. One common way to split identifiers is to follow programming language naming conventions. For example, Java programmers often use camel case, where words are delineated by uppercase letters or non-alphabetic characters. However, programmers also create identifiers by concatenating sequences of words together with no discernible delineation, which poses challenges to automatic identifier splitting.*

*In this paper, we present an algorithm to automatically split identifiers into sequences of words by mining word frequencies in source code. With these word frequencies, our identifier splitter uses a scoring technique to automatically select the most appropriate partitioning for an identifier. In an evaluation of over 8000 identifiers from open source Java programs, our Samurai approach outperforms the existing state of the art techniques.*

## 1. Introduction

Today's large, complex software systems require automatic software analysis and recommendation systems to help the software engineer complete maintenance tasks effectively and efficiently. The software maintainer must gain at least partial understanding of the concepts represented by, and the programmer's intent in, existing source code before making modifications. A programmer codes the

concepts and actions in terms of program structure, and helps to convey the intent and application domain concepts to human readers through identifier names and comments. Thus, many of the program search, concern location, code reuse, and quality assessment tools for software engineers are based on analyzing the words that programmers use in comments and identifiers.

Maintenance tools that analyze comments and identifiers frequently rely on first automatically identifying the individual words comprising the identifiers. Unlike natural languages, where space and punctuation are used to delineate words, identifiers cannot contain spaces. Often, programmers create identifiers with multiple words, called multi-word identifiers, to name the entity they want to represent (e.g., `toString`, `ASTVisitorTree`, `newValidatingXML-InputStream`, `jLabel6`, `buildXMLforComposite`).

To split multi-word identifiers, most existing automatic software analysis tools that use natural language information rely on coding conventions [1, 10, 11, 13, 14, 15, 20]. When simple coding conventions such as camel casing and non-alphabetic characters (e.g., '\_' and numbers) are used to separate words and abbreviations, automatically splitting multi-word identifiers into their constituent words is straightforward. However, there are cases where existing coding conventions break down (e.g., `DAYSforMONTH`, `GPSstate`, `SIMPLETYPE`NAME).

Techniques to automatically split multi-word identifiers into their constituent words can improve the effectiveness of a variety of natural language-based software maintenance tools. In program search, information retrieval (IR) techniques are preferred over regular expression based techniques because IR better captures the lexical concepts and can order the search results based on document relevance. However, IR techniques will miss occurrences of concepts if identifiers are not split. Similarly, incorrect splitting can decrease the accuracy of program search.

Consider searching for a feature that adds a text field to a report in a GUI-based report processing system. In the implementation, the developers are inconsistent with how

---

\*This material is based upon work supported by the National Science Foundation Grant No. CCF-0702401.

the concept of a “text field” is split. The methods responsible for adding the text field to the system uses proper camel case, `addTextField`. However, the GUI method responsible for initiating the action combines the concept as `textfield`. Thus, without correct identifier splitting, IR techniques will never return all relevant pieces of code for a single query (either “text field” or “textfield”). In this example, incorrect multi-word identifier splitting led to inaccuracy in the client software analysis because the words from incorrectly split identifiers misrepresent the source code’s lexical semantics.

Other tools that often rely on extracting individual concepts from the words used in the program are concern location [10, 11, 21, 22, 24, 26], documentation to source code traceability [1, 20, 25], or other software artifact analyses [2, 4, 5, 7, 12, 14, 23]. In addition to IR techniques, proper identifier splitting can help mine word relations from source code, by first extracting the correct individual words for analysis.

In this paper, we present a technique to automatically split identifiers into sequences of words by mining word frequencies in source code. Our novel identifier splitting algorithm automatically selects the most appropriate word partitioning for multi-word identifiers based on a scoring function. Because our approach requires no predefined dictionary, our word partitions are not limited by the words known to a manually created dictionary, and can naturally evolve over time as new words and technologies are added to the programmer’s vocabulary.

Our frequency-based approach, Samurai, is capable of correctly splitting multi-word identifiers involving camel case in which the straightforward split would be incorrect (e.g., `getWSstring`, `initPWforDB`, `ConvertASCIItoUTF`) and can also split same-case multi-words (e.g., `COUNTRYCODE`, `actionparameters`). We evaluate the effectiveness of our automatic identifier splitting technique by comparing mixed-case splitting with the only other automatic technique, which conservatively splits on division markers (e.g., camel casing, numbers and special characters), and by comparing same-case splitting with the state of the art automatic technique by Feild, Binkley, and Lawrie [8]. In an evaluation of over 8000 identifiers from open source Java programs, our Samurai approach outperforms the existing state of the art techniques. Although our current work focuses on Java programs predominantly written in English, our approach can be applied to any programming and natural language combination.

The major contributions of this paper are:

- Detailed analyses of the form of identifiers found in software and the challenges in automatically splitting them
- An effective technique for automatically splitting program identifiers into their constituent words

- An experimental evaluation comparing the accuracy of our frequency-based approach against the state-of-the-art identifier splitting approaches.

## 2. The Identifier (Token) Splitting Problem

Although the motivation for splitting arises from multi-word identifiers, splitting can also be applied to string literals and comments. In fact, identifiers frequently appear in Java doc comments as well as in sections of code that have been commented. Thus, in this paper, we focus on splitting *tokens*, which may be program identifiers or space-delimited strings appearing in code comments or string literals.

Token splitting is the problem of partitioning an arbitrary token into its constituent concept words, which are typically dictionary words and abbreviations. The general form of a token is a sequence of letters, digits, and special characters. In addition to using digits and special characters, another common convention for indicating word splits is camel casing [3, 6, 17, 19]. When using camel case, the first letter of every word in an identifier is capitalized (thus giving the identifier the look of a humped camel). Using capital letters to delimit words requires less typing than using special characters, while preserving readability. For example, `parseTable` is easier to read than `parsetable`.

Although initially used to improve readability, camel casing can also help to split tokens in static analysis tools that use lexical information. However, camel casing is not well-defined in certain situations, and may be modified to improve readability. Specifically, no convention exists for including acronyms within camel case tokens. For example, the whole abbreviation may be capitalized, as in `ConvertASCIItoUTF`, or just the first letter, as in `SqlList`. The decision depends on the readability of the token. In particular, `SqlList` is arguably more readable than `SQLList`, and more closely follows camel case guidelines than `SQLList`. Strict camel casing may be sacrificed for readability, especially for prepositions and conjunctions, as in `DAYSforMONTH`, `convertCEtoString`, or `PrintPError`. In some instances, no delimiters are used for very common multi-word concepts, such as `sizeof` or `hostname`. Thus, although camel case conventions exist, different decisions are made in the interest of readability and faster typing.

Since camel case tokens are so easy to parse, programmers may not be aware how common they are. In a 330 KLOC program with 44,315 tokens, we observed 32,817 multi-word tokens. Of this set, 1,856 tokens cannot be partitioned using straightforward camel case splitting, with 100 of the splits requiring alternating case (e.g., `DAYSforMONTH`) and 1,958 of the splits occurring within substrings of the same case (e.g., `sizeof`).

Formally, we define a token  $t = (s_0, s_1, s_3, \dots s_n)$ , where  $s_i$  is a letter, digit, or special character. The trivial first step in token splitting is to separate the token before and after each sequence of special characters and digits. Each substring is then considered as a candidate token to be further split. Any substrings left after the first trivial splits, we refer to as *alphabetic tokens*. An alphabetic token is a sequence of alternating upper and lower case letters. For example, `eof`, `Database`, `startCDATA`, `ConvertASCIItoUTF`, and `buildXMLforComposite` are all alphabetic tokens with varying numbers of alternation and sizes of substrings in a same-case sequence.

For alphabetic tokens, there are four possible cases to consider in deciding whether to split at a given point between  $s_i$  and  $s_j$ :

1.  $s_i$  is lower case and  $s_j$  is upper case (e.g., `getString`, `setPoint`)
2.  $s_i$  is upper case and  $s_j$  is lower case (e.g., `getMAXstring`, `GPSstate`, `ASTVisitor`)
3. both  $s_i$  and  $s_j$  are lower case (e.g., `notype`, `databasefield`, `actionparameters`)
4. both  $s_i$  and  $s_j$  are upper case (e.g., `USERLIB`, `NONNEGATIVEDECIMALTYPE`, `COUNTRYCODE`)

Case (1) is the natural place to split for straightforward camel case without abbreviations, (e.g., `isCellEditable`, `getDescription`). However, the examples for case (2) demonstrate how following strict camel casing can provide incorrect splitting (e.g., `getMA Xstring`, `GP Sstate`). We call the problem of deciding where to split when there is alternating lower and upper case present, the *mixed-case token splitting problem*. The mixed-case token splitting problem is particularly complicated by the use of acronyms.

We refer to cases (3) and (4) as the *same-case token splitting problem*. The programmer has not used any camel case, and thus has provided no clues as to whether any individual words, or concepts, should be extracted from the token.

A fully automatic program token splitting algorithm should automatically solve both the mixed-case and the same-case token splitting subproblems effectively. The algorithm should be capable of splitting a token into an arbitrary number of substrings that represent different concepts. The client software analysis tool can always merge together words that were split, but could be considered together as a single concept. For example, we observed that more experienced Java programmers would consider `javax` to be a single concept, the Javax API, while a novice would consider ‘java’ and ‘x’ to be separate words.

### 3. State of the Art

To our knowledge, Feild, Binkley, and Lawrie [8, 16, 18] are the only other researchers to develop and evaluate techniques that address the problem of automatically partitioning multi-word program identifiers. They define a string of characters between division markers (e.g., underscores and camel case) and the endpoints of a token to be a *hard word*. For example, the identifier `hashtable_entry` contains two hard words: `hashtable` and `entry`. When a hard word consists of multiple parts, the parts are called *soft words*. The hard word `hashtable` contains two soft words: `hash` and `table`. Thus, a hard word can consist of multiple soft words. Based on our understanding, a hard word containing more than one soft word is a same-case token.

Feild, et al. present two approaches to same-case token splitting—a greedy approach and an approach based on neural networks. The *greedy* approach uses a dictionary word list (from `ispell`), a list of known abbreviations, and a stop list of keywords which includes predefined identifiers, common library functions and variable names, and single letters. After returning each hard word found in one of the three word lists as a single soft word, the remaining hard words are considered for splitting. The algorithm recursively looks for the longest prefix and suffix that appear in one of the three lists. Whenever a substring is found in the lists, a division marker is placed at that position to signify a split and the algorithm continues. Thus, the greedy approach is based on a predefined dictionary of words and abbreviations, and splits are determined based on whether the word is found in the dictionary, with longer words preferred. In contrast, the *neural network* approach passes each hard word through a neural network to determine splits, with each network specialized to a given hard word length.

Feild, et al. evaluated the greedy and neural network approaches using a random algorithm for splitting as a base case. 4000 identifiers were randomly chosen from 746,345 identifiers in C, C++, Java, and Fortran codes. The hard words were determined by division markers, and the three techniques were run to identify soft words in each of the hard words. The neural network approach performed well when given specialized training data for a specific person, while the greedy algorithm was consistent across data sets. The greedy algorithm tended to insert more splits than desired.

Feild, et al. do not discuss the mixed-case token splitting problem, beyond stating that division markers are used to derive hard words. Our technique tackles the challenging situations in mixed-case token splitting that cannot be handled by the camel case rule of splitting before the alternation from upper to lower case. In contrast to the dictionary, length-based approach of Feild, et al., our approach uses the frequency of words both in the code and in a larger suite

of programs to score potential splits. We require no predefined dictionary, and automatically incorporate the evolving terminology so common in the field of software.

#### 4. Automatic Token Splitting with Samurai

Our automatic token splitting technique, called **Samurai**, is inspired by the mining approach to expanding programmer abbreviations, where the source code is mined for potential abbreviation expansions [9]. Our hypothesis is that the strings composing multi-word tokens in a given program are most likely used elsewhere in the same program, or in other programs. The words could have been used alone, or as part of another token. Our approach is also based on the hypothesis that the token splits that are more likely to represent the programmer’s intent are those splits that partition the token into strings that occur more often in the program. Thus, string frequency in the program is used to determine splits in the current token under analysis.

Samurai mines string frequency information from source code and builds two string frequency tables. We build both tables by first executing the conservative token splitting algorithm based on division markers (see section 5.1.1) on a set of source code tokens to generate a conservative listing of all occurrences of division-marker delimited strings in the source code tokens, which we call the extracted string list. The extracted string list is then mined for unique string frequency information, in the form of a lookup table that stores the number of occurrences of each unique string. A *program-specific frequency table* is built by mining over the strings extracted for the program under analysis. A *global frequency table* is built by mining over the set of strings extracted from a large corpus of programs. The two frequency tables are used in the scoring function that Samurai applies to a given string during the token splitting process.

For each token being analyzed, Samurai token splitting starts by executing the *mixedCaseSplit* algorithm. Shown in Algorithm 1, *mixedCaseSplit* outputs a space-delimited token where each space-delimited string of letters takes the form: (a) all lower case, (b) all upper case, or (c) a single upper case followed by all lower case letters. The output token is then processed by the *sameCaseSplit* algorithm, shown in Algorithm 2, which outputs a space-delimited token in which some of its substrings have been further split and delimited by spaces. Each split is denoted by an inserted blank character, and the final split token will be a sequence of substrings of the original token with an inserted blank character at each split. The following subsections describe each of the algorithms in detail as well as the scoring function.

---

#### Algorithm 1 *mixedCaseSplit(token)*

---

```

1: Input: token to be split, token
2: Output: space-delimited split token, sToken
3:
4: token = splitOnSpecialCharsAndDigits(token)
5: token = splitOnLowercaseToUppercase(token)
6: sToken ← ""
7:
8: for all space-delimited substrings s in token do
9:
10:   if  $\exists \{ i \mid \text{isUpper}(s[i]) \wedge \text{isLower}(s[i + 1]) \}$  then
11:
12:     n ← length(s) - 1
13:
14:     // compute score for camelcase split
15:     if i > 0 then
16:       camelScore ← score(s[i, n])
17:     else
18:       camelScore ← score(s[0, n])
19:     end if
20:
21:     // compute score for alternate split
22:     altScore ← score(s[i + 1, n])
23:
24:     // select split based on score
25:     if camelScore >  $\sqrt{\text{altScore}}$  then
26:       if i > 0 then
27:         s ← s[0, i - 1] + " " + s[i, n]
28:       end if
29:     else
30:       s ← s[0, i] + " " + s[i + 1, n]
31:     end if
32:
33:   end if
34:
35:   sToken ← sToken + " " + s
36:
37: end for
38:
39: token ← sToken
40: sToken ← ""
41:
42: for all space-delimited substrings s in token do
43:   sToken ← sToken + " " + sameCaseSplit(s,
44:     score(s)
45:   end for
46: return sToken

```

---

## 4.1. Mixed-case Token Splitting

The *mixedCaseSplit* algorithm begins by replacing special characters with blank characters and inserting a blank character before and after each sequence of digits. The *splitOnLowercaseToUpper* function adds a blank character between every two-character sequence of a lower case letter followed by an upper case letter. At this point, each alphabetic substring is of the form zero or more upper case letters followed by zero or more lowercase characters (e.g., List, ASTVisitor, GPSstate, state, finalstate, NAMESPACE, MAX).

Each mixed-case alphabetic substring is then examined to decide between the straightforward camel case splitting before the last upper case letter (e.g., “AST Visitor”, “GP Sstate”) or the alternate split between the last upper case letter and the first lower case letter (e.g., “ASTV isitor”, “GPS state”). The split selection is determined by comparing the score of the string to the right of the split point, dampening the alternate split score to favor the camel case split unless there is overwhelming evidence for the alternate split. The original alphabetic substring is replaced in the token by the split substring.

After all mixed-case splitting is completed, the *mixedCaseSplit* algorithm calls the *sameCaseSplit* algorithm on each space-delimited substring of the current (possibly already split) token. The string score of the space-delimited substring is input to the *sameCaseSplit* to be used in making split decisions, particularly decisions in recursive calls to *sameCaseSplit*. The substrings returned from *sameCaseSplit* are concatenated with space delimiters to construct the final split token.

## 4.2. Same-case Token Splitting

Along with the score of the original same-case token to be split, the *sameCaseSplit* algorithm takes as input the substring under analysis,  $s$ , which is already (1) all lower case, (2) all upper case, or (3) a single upper case letter followed by all lower case letters. Starting with the first position in  $s$ , the algorithm examines each possible split point in  $s$ , where  $s$  is split into two substrings, called *left* and *right*, with  $\text{score}(\textit{left})$  and  $\text{score}(\textit{right})$ , respectively.

The split decision is based on several conditions. Intuitively, we are looking for a split with the largest possible score such that (a) the substrings are not common prefixes or suffixes and (b) there is overwhelming evidence to support the split decision. Based on exploratory data analysis, we determined “overwhelming evidence in favor of a split” to be when a dampened score for each of the potential substrings is larger than both the  $\text{score}(s)$  and the score of the original same-case string (before any recursive *sameCaseSplit* calls). If these conditions hold,  $s$  will be

---

### Algorithm 2 *sameCaseSplit*( $s, \text{score}_{ns}$ )

---

```

1: Input: same-case string,  $s$ 
2: Input: no split score,  $\text{score}_{ns}$ 
3: Output: final space-delimited split token,  $\textit{splitS}$ 
4:
5:  $\textit{splitS} \leftarrow s, n \leftarrow \text{length}(s) - 1$ 
6:  $i \leftarrow 0, \text{maxScore} \leftarrow -1$ 
7:
8: while  $i < n$  do
9:    $\text{score}_l \leftarrow \text{score}(s[0, i])$ 
10:   $\text{score}_r \leftarrow \text{score}(s[i + 1, n])$ 
11:   $\textit{prefix} \leftarrow \text{isPrefix}(s[0, i]) \vee \text{isSuffix}(s[i + 1, n])$ 
12:   $\textit{toSplit}_l \leftarrow \sqrt{\text{score}_l} > \max(\text{score}(s), \text{score}_{ns})$ 
13:   $\textit{toSplit}_r \leftarrow \sqrt{\text{score}_r} > \max(\text{score}(s), \text{score}_{ns})$ 
14:  if  $\neg \textit{prefix} \wedge \textit{toSplit}_l \wedge \textit{toSplit}_r$  then
15:    if  $(\text{score}_l + \text{score}_r) > \text{maxScore}$  then
16:       $\text{maxScore} \leftarrow \text{score}_l + \text{score}_r$ 
17:       $\textit{splitS} \leftarrow s[0, i] + " " + s[i + 1, n]$ 
18:    end if
19:  else if  $\neg \textit{prefix} \wedge \textit{toSplit}_l$  then
20:     $\textit{temp} \leftarrow \text{sameCaseSplit}(s[i + 1, n], \text{score}_{ns})$ 
21:    if  $\textit{temp}$  was further split then
22:       $\textit{splitS} \leftarrow s[0, i] + " " + \textit{temp}$ 
23:    end if
24:  end if
25:   $i \leftarrow i + 1$ 
26: end while
27: return  $\textit{splitS}$ 

```

---

split and no more splitting of  $s$  will be attempted. If condition (a) holds but only the substring to the left of the current potential split point (*left*) provides overwhelming evidence of being a word, then *sameCaseSplit* is called recursively to determine whether *right* should be further split. If *right* results in being split, then we split between *left* and *right* also, otherwise we do not split here because data analysis showed that splitting at the current point based solely on the evidence of *left* would result in improper splits (e.g., “string ified”). Following this recursive algorithm, Samurai correctly splits nonnegativedecimaltype as “nonnegative decimal type”.

One challenge we faced in developing a frequency-based token splitting technique is that short words occur so frequently in the source code that they tend to have much higher scores than longer words. If the algorithm does not include some way to dampen the scores of short words, tokens will be split incorrectly by splitting words that should not be split. This led us to perform the square root of the substring scores before comparing them to the scores of the current and original same-case strings being analyzed. Otherwise, splits would be improperly inserted with short words, when a better split existed (e.g., performed would

be split as “per formed”).

### 4.3 String Scoring Function

A key component of our token splitting technique is our string scoring function,  $score(s)$ , that returns a score for the string  $s$  based on how frequently  $s$  appears in the program under analysis and in a more global scope of a large set of programs. The  $score$  function is called numerous times particularly to make decisions at two key steps in our automatic token splitting algorithms:

- to score and compare the straightforward camel case split results to the alternate split results during mixed-case token splitting
- to score different substring partitions of same-case tokens to identify the best split

We compute the string score for a given string  $s$  by computing the following function:

$$Freq(s, p) + (globalFreq(s) / \log_{10}(AllStrsFreq(p)))$$

where  $p$  is the program under analysis.  $Freq(s, p)$  is the frequency that string  $s$  occurs in the program  $p$ .  $globalFreq(s)$  is the frequency that string  $s$  occurs in a large set of Java programs.  $AllStrsFreq(p)$  is the total frequency of all strings in the program  $p$ .

As mentioned earlier, the frequency information is mined from the source code. Over 9000 Java SourceForge projects were mined for the occurrences of strings to create the global frequency table used in  $globalFreq(s)$ . Approximately 630,000 unique conservatively split words were identified with a total number of occurrences of 938 million. The number of occurrences of strings in a given program varies with the program size. Larger programs have a higher frequency of string occurrences. The frequency analysis runs in linear time with respect to the number of identifiers in the program.

The formula for the scoring function was developed through exploratory data analysis. Sometimes the mined program-specific frequency table has high frequency for a string that does not appear with high frequency in the larger program corpus. However, for small programs, the frequency of a string in the program itself may be too low to provide good data for splitting, motivating the inclusion of the mined global frequency table information. Thus, the string scoring function is comprised of both the frequency of the string in the program under analysis as well as the global frequency of strings.

However, the frequency of a string within a program can be overly dominated by the global frequency of that string due to the much larger mined data set from 9000 programs, when compared to the relatively small amount of

data mined from a single program. Thus, we dampen the effect of the global frequency in the score by dividing by the log of the frequency of all strings in the program under analysis. This takes into consideration the fact that we have more mined frequency data from larger programs than smaller programs.

### 4.4 Implementation

Our token splitting technique is fully automatic and implemented as a standalone tool composed of a set of perl scripts. Samurai takes two lists of tokens from Java programs as input—the tokens from the program under analysis and the global list of tokens from the SourceForge programs, along with the set of tokens to be split. It outputs the set of split tokens, split by spaces. The current implementation is designed for batch processing of a set of tokens, but could be incrementally updated or run in the background to support software maintenance tools. Samurai uses a list of common prefixes and suffixes (which are available online: <http://www.cis.udel.edu/~enslen/samurai>).

## 5. Evaluation

We evaluated our automatic token splitting technique with two research questions in mind:

1. How does our technique for mixed-case token splitting compare with the only other automatic technique, conservative division marker (e.g., camel case and special character) splitting?
2. How does our technique for same-case token splitting compare with the state of the art Feild, Binkley, and Lawrie [8] greedy approach?

### 5.1. Experiment Design

#### 5.1.1 Variables and Measures

The independent variable is the token splitting algorithm, which we evaluate by measuring the accuracy of each technique in identifying the correct partitioning of a gold set of tokens from Java programs.

To evaluate how effectively our mixed-case token splitting performs, we implemented the straightforward splitting based on division markers and camel casing used by most researchers in software analysis who require token splitting, which we call *conserv*. Our implementation scans a token and splits between any alternating case from lower case to upper case and before the last upper case in a sequence of multiple consecutive upper case followed by lower case, between letters and digits, and treating special characters as

delimiters. For simple camel case tokens, such as `getString` or `setIdIdentifierType`, the tokens would be correctly split as “get String” and “set Identifier Type”. For strings such as `DAYSforMONTH`, this conservative camel case approach will incorrectly split as “DAY Sfor MONTH”. However, the conservative camel case approach does not require any knowledge of abbreviations or common situations that do not follow this simple rule, thus making it very efficient and easy to implement.

To evaluate the effectiveness of our same-case token splitting, we implemented the greedy algorithm by Feild, Binkley, and Lawrie [8]. For the three predefined lists, we used the same `ispell` Version 3.1.20, a stop list of Java keywords, and an abbreviation list that we had created as part of our earlier work on abbreviation expansion [9]. Because we are evaluating with Java programs, we did not include library functions and predefined identifiers in the stop list. We did not compare with the neural network approach because it was only shown to perform well given specialized data, while the greedy approach was more consistent across data sets. Since they did not have an automatic front-end for splitting mixed-case, we ran conservative camel case for the mixed-case splitting.

The dependent variable in our study is the effectiveness of each technique, measured in terms of accuracy. Accuracy is determined by counting the number of tokens that are correctly split, where correctly split is defined to be completely matching splits with the human annotators who produced the gold set. Tokens with multiple splits, but only some of them correctly split by an automatic technique, were considered to be incorrectly split. Because not all trends in the data are visible in terms of accuracy, we also measured correctness in terms of the percent of incorrectly split same-case tokens, or *oversplitting*.

### 5.1.2 Subjects

We randomly selected tokens from 9000 open source Java programs in SourceForge. Two human annotators who had no knowledge of our token splitting technique manually inspected each token in their respective sets to identify the most appropriate token splits. To construct the gold set, we continued to add tokens to each human subject’s set until we reached 1500 nondictionary words in their set. The total number of tokens in the gold set is 8466.

### 5.1.3 Methodology

We mined each of the 9000 programs to produce the program-specific and global frequency tables. We ran four techniques on the entire set of tokens in the gold set: conservative camel case, Samurai, greedy, and *mixedCaseSplit* without the call to *sameCaseSplit*. We compared the output

of each tool with the gold set. If the space-delimited token generated by the automated technique was identical to the human-split token, then the automatic token split is considered to be correct. We computed the accuracy for each tool. We then computed accuracy for different categories of tokens based on their characteristics to analyze the differences in effectiveness. We also computed the amounts of oversplitting performed by Samurai and greedy.

## 5.2. Threats to Validity

Because our technique is developed on Java programs, the results of the study may not generalize to all programming languages; however, the gold set does include tokens in natural languages other than English.

As with any subjective task, it is possible that the human annotators did not identify the correct split for a given token. In some instances, the annotators kept proper names together even when they were camel cased. There were a number of same-case tokens that were ambiguous, and up to personal preference. We noticed that subjects more familiar with Java programming would split differently from a novice programmer. For instance, the splitting of `Javadoc`, `sourceforge`, `gmail`, `gcal` are subjective.

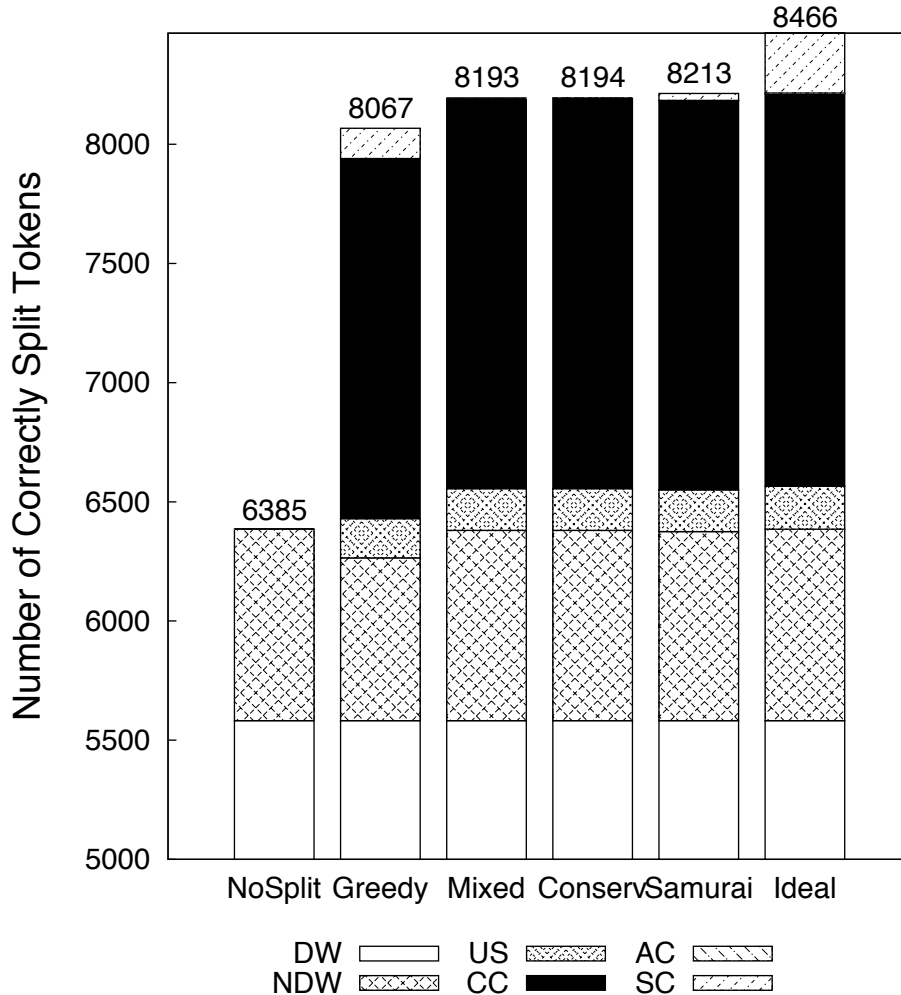
Because we did not have access to the actual lists used in the greedy algorithm, we tried to be as fair as possible in our implementation of greedy by using the same dictionary that they referenced, a list of common abbreviations from our previous work [9], and a stop list of common words in Java.

## 6. Results and Analysis

We present the accuracy results for our experiment in Figure 1. Although Samurai misses some same-case splitting, Samurai is more accurate than the greedy algorithm overall.

### 6.1. Mixed-case

Samurai performs very similar to *conserv* in mixed-case splitting. There are 1632 instances where the split is camel case. Samurai correctly chooses the camel case split in 1630 cases, and incorrectly chooses the alternate split in just two cases. There are only four instances of alternate split, one of which Samurai correctly selects the alternate split. *conserv* does not get any of these correct because it never considers alternating splits. Thus, for mixed-case splitting, even this large data set is not a large enough sample to answer this question. To get a realistic data sample, we sacrificed targeting specific types of mixed-case tokens. In the future, we



**Figure 1. Accuracy of token splitting approaches per category: dictionary word requiring no split (DW), nondictionary word without split (NDW), underscores and digits (US), camel case (CC), alternating case (AC), same-case (SC).**

plan to perform a more targeted evaluation toward mixed-case splitting.

### 6.2. Same-case

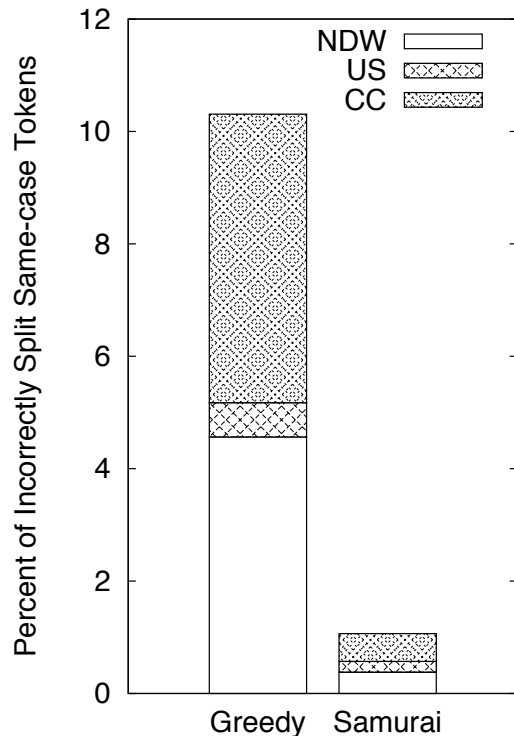
Note that in Figure 1, except for greedy, the techniques correctly do not split dictionary and nondictionary tokens. Also, note that the greedy algorithm outperforms Samurai in terms of same-case splitting. There are 249 tokens that contain at least one same-case split. Greedy correctly splits 125 of these tokens, while Samurai correctly splits just 29.

Although Samurai splits fewer of the same-case tokens that should be split, it makes fewer mistakes than greedy in splitting strings that should not be split. There are 6391 *no-split* tokens, which require no split. 5582 of these no-

split tokens are dictionary words, and 809 are nondictionary tokens. Because greedy uses the same dictionary, it is no surprise that greedy does not split the dictionary words. It should be noted that even though Samurai does not use a dictionary, it also does not split any of the dictionary words.

The results for no-split nondictionary tokens are not as favorable for greedy. Figure 2 presents the results on the percent of oversplitting by greedy and Samurai. The incorrectly split same-case words are full nondictionary tokens, words that fall between underscores/digits, or camel case separated words. As expected from FBL’s evaluation [8], greedy suffers from a significant amount of oversplitting. In contrast to greedy’s 10%, Samurai’s frequency-based approach oversplits in just 1% of cases.

Although Samurai splits fewer same-case tokens than



**Figure 2. Percent of incorrectly split same-case tokens by category: nondictionary word (NDW), underscore/digit separated words (US), camel case separated words (CC).**

greedy, it is more accurate overall by oversplitting significantly less. The data reveal that our scoring function may be overly conservative. In future, we plan to investigate a scoring function that more accurately balances splitting same-case tokens while preserving no-split tokens.

## 7. Conclusion

Automatically partitioned tokens can be used to increase the accuracy of natural language analysis of software, which is a key component of many software maintenance tools including program search, concern location, documentation to source traceability, and software artifact analyses. In this paper, we presented and evaluated a technique to automatically split tokens into sequences of words by mining the frequency of potential substrings from the source code.

We evaluated Samurai against the state of the art on over 8000 tokens. Our results show that frequency-based token splitting misses same-case splits identified by the greedy algorithm, but outperforms greedy overall by making significantly fewer oversplits. Samurai also identifies slightly

more correct splits than conservative division marker splitting, without incorrectly splitting any dictionary words.

The Samurai technique could be further improved by tweaking the conditions for the scoring function with respect to same-case splitting. In addition, improvements could be made by determining when to merge words back together, especially those with digits (e.g., MP3), to include both the split and merged form in the final set. In the future, we plan to evaluate our approach combined with abbreviation expansion [9].

## 8. Acknowledgments

The authors would like to thank Haley Boyd, Amy Siu, and Sola Johnson for their invaluable help on this paper.

## References

- [1] G. Antoniol, G. Canfora, G. Casazza, A. D. Lucia, and E. Merlo. Recovering traceability links between code and documentation. *IEEE Transactions on Software Engineering*, 28(10):970–983, 2002.
- [2] J. Anvik, L. Hiew, and G. C. Murphy. Who should fix this bug? In *ICSE '06: Proceeding of the 28th international conference on Software engineering*, pages 361–370, New York, NY, USA, 2006. ACM Press.
- [3] B. Caprile and P. Tonella. Nomen est omen: Analyzing the language of function identifiers. In *WCRE '99: Proceedings of the 6th Working Conference on Reverse Engineering*, pages 112–122, 1999.
- [4] D. Čubranić and G. C. Murphy. Hipikat: recommending pertinent software development artifacts. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 408–418, 2003.
- [5] D. Čubranić and G. C. Murphy. Automatic bug triage using text classification. In *In Proceedings of Software Engineering and Knowledge Engineering*, pages 92–97, 2004.
- [6] F. Deissenboeck and M. Pizka. Concise and consistent naming. *Software Quality Control*, 14(3):261–282, 2006.
- [7] G. di Lucca. An approach to classify software maintenance requests. In *ICSM '02: Proceedings of the International Conference on Software Maintenance (ICSM'02)*, page 93, Washington, DC, USA, 2002. IEEE Computer Society.
- [8] H. Feild, D. Binkley, and D. Lawrie. An empirical comparison of techniques for extracting concept abbreviations from identifiers. In *Proceedings of IASTED International Conference on Software Engineering and Applications (SEA'06)*, Nov. 2006.
- [9] E. Hill, Z. P. Fry, H. Boyd, G. Sridhara, Y. Novikova, L. Pollock, and K. Vijay-Shanker. AMAP: Automatically mining abbreviation expansions in programs to enhance software maintenance tools. In *MSR '08: Proceedings of the Fifth International Working Conference on Mining Software Repositories*, Washington, DC, USA, 2008. IEEE Computer Society.

- [10] E. Hill, L. Pollock, and K. Vijay-Shanker. Exploring the neighborhood with Dora to expedite software maintenance. In *ASE '07: Proceedings of the 22nd IEEE International Conference on Automated Software Engineering (ASE'07)*, pages 14–23, Washington, DC, USA, November 2007. IEEE Computer Society.
- [11] E. Hill, L. Pollock, and K. Vijay-Shanker. Automatically capturing source code context of nl-queries for software maintenance and reuse. In *ICSE '09: Proceedings of the 31st international conference on Software engineering*, 2009.
- [12] P. Hooimeijer and W. Weimer. Modeling bug report quality. In *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 34–43, New York, NY, USA, 2007. ACM.
- [13] E. W. Host and B. M. Ostvold. The programmer's lexicon, volume I: The verbs. In *SCAM '07: Proceedings of the Seventh IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 193–202, Washington, DC, USA, 2007. IEEE Computer Society.
- [14] M. Kim, D. Notkin, and D. Grossman. Automatic inference of structural changes for matching across program versions. In *ICSE '07: Proceedings of the 29th international conference on Software Engineering*, pages 333–343, Washington, DC, USA, 2007. IEEE Computer Society.
- [15] A. Kuhn, S. Ducasse, and T. Girba. Semantic clustering: Identifying topics in source code. *Information Systems and Technologies*, 49(3):230–243, 2007.
- [16] D. Lawrie, H. Feild, and D. Binkley. An empirical study of rules for well-formed identifiers: Research articles. *J. Softw. Maint. Evol.*, 19(4):205–229, 2007.
- [17] D. Lawrie, H. Feild, and D. Binkley. Extracting meaning from abbreviated identifiers. In *SCAM '07: Proceedings of the 7th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2007)*, pages 213–222, 2007.
- [18] D. Lawrie, H. Feild, and D. Binkley. Quantifying identifier quality: an analysis of trends. *Empirical Softw. Engg.*, 12(4):359–388, 2007.
- [19] B. Liblit, A. Begel, and E. Sweetser. Cognitive perspectives on the role of naming in computer programs. In *Proceedings of the 18th Annual Psychology of Programming Workshop*, 2006.
- [20] A. Marcus and J. I. Maletic. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 125–135, 2003.
- [21] A. Marcus, A. Sergeyev, V. Rajlich, and J. I. Maletic. An information retrieval approach to concept location in source code. In *WCRE '04: Proceedings of the 11th Working Conference on Reverse Engineering (WCRE'04)*, pages 214–223, 2004.
- [22] D. Poshyvanyk, M. Petrenko, A. Marcus, X. Xie, and D. Liu. Source code exploration with Google. In *ICSM '06: Proceedings of the 22nd IEEE International Conference on Software Maintenance (ICSM'06)*, pages 334–338, 2006.
- [23] P. Runeson, M. Alexandersson, and O. Nyholm. Detection of duplicate defect reports using natural language processing. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, pages 499–510, Washington, DC, USA, 2007. IEEE Computer Society.
- [24] D. Shepherd, Z. P. Fry, E. Hill, L. Pollock, and K. Vijay-Shanker. Using natural language program analysis to locate and understand action-oriented concerns. In *AOSD '07: Proceedings of the 6th International Conference on Aspect-oriented Software Development*, 2007.
- [25] S. Yadla, J. Huffman Hayes, and A. Dekhtyar. Tracing requirements to defect reports: an application of information retrieval techniques. *Innovations in Systems and Software Engineering*, 1(2):116–124, 2005.
- [26] W. Zhao, L. Zhang, Y. Liu, J. Sun, and F. Yang. SNI-AFL: Towards a static non-interactive approach to feature location. *ACM Transactions on Software Engineering and Methodology*, 15(2):195–226, 2006.