

# An Eclipse Plug-in for Enforcing Java Naming Conventions

Matthew Strand<sup>1\*</sup>, Lindsey Hughes<sup>1\*</sup>, Robert Duncan<sup>2\*</sup>, Joe Smith<sup>1</sup>, Erik Linstead<sup>1</sup>

<sup>1</sup>Department of Math and Computer Science, Chapman University.

<sup>2</sup>Department of Informatics, University of California, Irvine.

stran104@chapman.edu, hughe120@chapman.edu, rduncan@uci.edu, smith237@chapman.edu, linstead@chapman.edu

## ABSTRACT

We present a preliminary version of an Eclipse plug-in to assist novice Java programmers in adhering to standard naming conventions and software vocabulary. Our plug-in leverages first-order Markov chains trained on 12,151 open source Java projects to model heuristics for class, interface, method, and field identifiers. When integrated with the Eclipse framework these statistical machine learning techniques provide an effective and novel tool for new programmers to analyze their code in order to identify and correct deviations from the common naming practices followed by the software engineering community.

## 1. INTRODUCTION

In recent years the increasing availability of publicly available open source code has paved the way for the application of machine learning and information retrieval techniques to the software engineering domain. The application of these techniques are as varied as the methods themselves, from mining code concepts with latent semantic indexing [3], to improving code search with graph algorithms [2], to facilitating debugging with unsupervised text analysis [1]. Though these tools share the goal of improving developer productivity and efficiency in general, few of these data mining applications are targeted to the audience that could perhaps most benefit from their use - novice programmers.

In this poster we make further progress in applying machine learning to computer science education by describing the preliminary version of an Eclipse plug-in for assessing Java source code for adherence to the naming conventions and vocabulary mandated by software engineering best practices, with the end goal of improving the readability and maintainability of code developed by new programmers. This goal is achieved by adapting first-order Markov

chains to probabilistically model the structure and vocabulary of class, interface, method, and field identifiers in the Java programming language. Our models are based on the analysis of 12,151 projects indexed by the Sourcerer software repository [4], and provide a statistically significant basis for assisting new Java developers in locating poor naming conventions in their code while being able to benefit from the full-featured integrated development environment (IDE) provided by Eclipse.

## 2. METHODOLOGY

In order to train the Markov models that form the heart of our Eclipse plug-in we start with the fully qualified names (FQN's) stored in Sourcerer. From each FQN we extract the words that constitute the entity name, where the entity name is the rightmost portion of the FQN. Once the proper entity name is obtained, it is tokenized to derive individual vocabulary words by applying common heuristics. Parsing our full Java corpus yields a vocabulary of 44,060 distinct words. When examined individually, classes boast a vocabulary size of 17,286, interfaces a size of 5,091, methods a size of 23,699 and fields a vocabulary of 31,231 words.

As part of our processing we store the sequence of individual words that comprise each entity name, which then forms the training data for our Markov Models. The sequence  $S = (w_0, \dots, w_k, \dots, w_K)$  is said to be a first-order Markov chain if  $\Pr(w_{k+1}|w_k, \dots, w_0) = \Pr(w_{k+1}|w_k)$ , meaning that given word  $w_k$  the word  $w_{k+1}$  is conditionally independent of all previous words in  $S$ . Informally, at a given position in the word sequence, the next word is probabilistically determined only by the current word, and not by past words.

For each entity type a first-order Markov Model can be trained by counting the frequency of each bigram in the naming sequence data, as well as the frequency of the first words in the sequences. These counts are then normalized into probability distributions. The result is a  $V \times V$  transition table (where  $V$  is the vocabulary size) such that entry  $(i, j)$  is the probability that word  $j$  follows word  $i$ . Bigrams that are not observed in the training data may be accounted for using techniques such as Laplace smoothing or Good-Turing discounting. Once a model is trained for an entity type, the probability of a new sequence,  $S'$ , being generated from the model can be calculated using basic statistics. Trained models may be persisted using any number of sparse matrix schemes, either in a flat file or relational database, and may be updated in an incremental fashion.

\* These authors contributed equally.

### 3. PLUG-IN OPERATION

The architecture of our Eclipse plug-in consists of three main components: (1) The Markov Models described in section 2, (2) A code parser and identifier tokenizer, (3) A user interface for reporting deviations on naming conventions to the user. While space constraints prevent inclusion of detailed design artifacts and screenshots, these will be included in the poster.

When the user invokes our plug-in from the Eclipse IDE the current Java source file is first parsed to obtain a list of class, interface, method, and field names. Each of these names are then tokenized using standard heuristics, such as splitting on underscores and case changes. The result is a sequence of words for each program entity that is assessed for naming convention adherence via the trained Markov Models.

Identifying naming conventions that deviate from best-practice can be reduced to a classification problem based on the probability that a sequence of tokens was generated from a class, interface, method, or field entity. To this end, each sequence that is extracted from the processed source file is scored against each of the four underlying Markov Models trained specifically for our plug-in. The probability that a sequence  $S = (w_0, \dots, w_k, \dots, w_K)$  is a class, for example, is given by

$$\Pr(class|S) = \frac{\Pr(S_0 = w_0|class) \prod_{k=1}^K \Pr(w_k|w_{k-1}) \Pr(class)}{\Pr(S)}$$

where  $\Pr(class)$  is the prior probability on classes. The probability that  $S$  is an interface, method, or field can be calculated by making the appropriate substitutions into the equation above. Because the evidence term  $\Pr(S)$  is the same in all cases it effectively serves as a normalizing constant that can be ignored in practice.

Once scores have been computed for all types, the sequence under analysis is classified as the most probable entity,  $\hat{E}$ . This predicted entity label is then compared to the actual entity label of the sequence,  $E$ , as determined from parsing the source file. If the predicted entity label and the actual entity label are the same ( $E = \hat{E}$ ), then the identifier under consideration is deemed to adhere to naming conventions for its entity type based on the probabilistic evidence encapsulated in the Markov Model. If the predicted entity label and the true entity label do not agree ( $E \neq \hat{E}$ ) then the identifier is determined to deviate from naming conventions for its type. In this case an error message is displayed to the user in the Eclipse user interface containing the line of the deviation, the most likely type of the identifier given its name, and the probability that the identifier is of the predicted type.

The utility of the plug-in is perhaps best demonstrated with a simple example. Consider a novice programmer who has developed a single class named “CalculateGrade” which contains a single method “gradeCalculator.” Both the class and method names exhibit an obvious deviation from naming convention, as methods typically start with verbs [5]. When the class and method names are processed by our plug-in it will report, with a high probability, that “CalculateGrade” is more appropriate for a method name than a class name,

and that “gradeCalculator” is more appropriate for a class name rather than a method name. The user can then update their implementation accordingly, with one solution being that the entity names are simply swapped. When the plug-in is run again the underlying models will verify that indeed “GradeCalculator” is a suitable class name, and the same for the method “calculateGrade,” and no errors will be reported to the user.

### 4. RELATED AND FUTURE WORK

In our previous work we have conducted an in-depth study of Java software vocabulary [6], as well as the performance of first-order Markov models for the purposes of entity classification using FQN’s [5]. The reader may refer to these publications for a complete literature review of modelling software naming conventions. Our previous findings form the theoretical basis for the Eclipse plug-in described here, but made no progress toward developing a tool to assist novice programmers in learning effective naming conventions. We are currently in the process of improving and expanding the preliminary version of our plug-in. We are developing higher-order Markov chains for the underlying models. To simplify distribution and deployment we are migrating our classifiers to web services backed by relational databases, rather than the locally persisted models that require several hundred megabytes of disk space. Finally, we are updating the user interface with features for highlighting and investigating deviant naming conventions found by the plug-in in the developers code.

### 5. REFERENCES

- [1] D. Andrzejewski, A. Mulhern, B. Liblit, and X. Zhu. Statistical debugging using latent topic models. In S. Matwin and D. Mladenic, editors, *18th European Conference on Machine Learning*, Warsaw, Poland, Sept. 17–21 2007. To appear.
- [2] K. Inoue, R. Yokomori, H. Fujiwara, T. Yamamoto, M. Matsushita, and S. Kusumoto. Component rank: relative significance rank for software component search. In *ICSE ’03: Proceedings of the 25th International Conference on Software Engineering*, pages 14–24, Washington, DC, USA, 2003. IEEE Computer Society.
- [3] A. Kuhn, S. Ducasse, and T. Girba. Semantic clustering: Identifying topics in source code. *Information and Software Technology (to appear)*, 2006.
- [4] E. Linstead, S. Bajracharya, T. Ngo, P. Rigor, C. Lopes, and P. Baldi. Sourcerer: mining and searching internet-scale software repositories. *Data Min. Knowl. Discov.*, 18(2):300–336, 2009.
- [5] E. Linstead, L. Hughes, C. Lopes, and P. Baldi. Capturing java naming conventions with first-order markov models. In *Proc. International Conference on Program Comprehension*, pages 313–314, May 2009.
- [6] E. Linstead, L. Hughes, C. Lopes, and P. Baldi. Exploring java software vocabulary: A search and mining perspective. In *SUITE ’09: Proceedings of the 2009 ICSE Workshop on Search-Driven Development-Users, Infrastructure, Tools and Evaluation*, pages 29–32, Washington, DC, USA, 2009. IEEE Computer Society.