Code Template Inference Using Language Models

Ferosh Jacob and Robert Tairas
University of Alabama at Birmingham
Computer and Information Sciences
Birmingham, Alabama
jacobf, tairasr@cis.uab.edu

ABSTRACT

This paper investigates the use of a natural language processing technique that automatically detects project-specific code templates (i.e., frequently used code blocks), which can be made available to software developers within an integrated development environment. During software development, programmers often and in some cases unknowingly rewrite the same code block that represents some functionality. These frequently used code blocks can inform the existence and possible use of code templates. Many existing code editors support code templates, but programmers are expected to manually define these templates and subsequently add them as templates in the editor. Furthermore, the support of editors to provide templates based on the editing context is still limited. The use of n-gram language models within the context of software development is described and evaluated to overcome these restrictions. The technique can search for project-specific code templates and present these templates to the programmer based on the current editing context.

Categories and Subject Descriptors

D.2.3 [**Software Engineering**]: Coding Tools and Techniques—*Program editors*; I.2.7 [**Artificial Intelligence**]: Natural Language Processing—*Language generation, Language models, Language parsing and understanding, Text analysis*

General Terms

Design

Keywords

template, source code, auto complete, n-gram language model

1. INTRODUCTION

Programmers often rewrite or use the same block of code in multiple locations (i.e., code clones) in the source code. Code templates represent some form of program logic that can be customized and potentially be used in other contexts. With many source code editors providing support for code templates, programmers can utilize templates during coding. Instead of starting from scratch, a template for a specific functionality can be inserted in the source code and subsequently customized by the programmer. However, the template support in these editors requires the programmer to manually determine which code blocks should be made templates and also manually add these templates within the editor. These tasks can potentially become tedious and time consuming for the programmer. In addition, suggesting code templates based on the editing context in which the programmer is currently working is still limited. Editors are unable to suggest a template based on the current programming language construct that is being edited. In Eclipse[1], every template is linked with a key and the programmer is expected to remember the key to make use of that template. In this paper, a technique with tool support is introduced that fetches templates for a given input string using the n-gram language model, which can predict the most likely item following the $n-1$ items in the history by training on previously identified templates. The suggested templates are displayed to the user based on the current editing context and hence can be provided during the actual editing of the source code.

The rest of the paper is structured as follows: the next section provides a motivating scenario and outlines the technique that is used. Section 3 provides implementation details of the proof of concept. Section 4 evaluates and compares the tool support for the approach discussed in this paper with current techniques for dealing with code templates. Related work that includes other ventures of applying machine learning (ML) techniques in software engineering are outlined in Section 5. The paper concludes with a description of possible extensions to strengthen and widen the scope of the project.

2. MOTIVATING SCENARIO

Consider a programmer attempting to use the API for Hibernate[2], which maps Java classes to database tables to allow persistence of objects. In order to properly use the API commands provided by Hibernate, a programmer must know the sequence of statements that is needed to perform a specific function. The programmer can refer to the API's documentation (i.e., Javadoc), but he or she must manually deter-

[1]http://www.eclipse.org
[2]http://www.hibernate.org

ACMSE '10 April 15-17, 2010, Oxford, MS, USA

```
26 Session secondSession = HibernateUtil
27                         .getSessionFactory()
28                         .openSession();
29 Transaction secondTransaction = secondSession.beginTransaction();
30 List messages =secondSession
31                 .createQuery("from Message where chat_id='"+chat_id+"'")
32                 .list();
33 System.out.println( messages.size() + " message(s) found:" );
34 secondTransaction.commit();
35 secondSession.close();
36 return messages;
37
```

Figure 1: Sample code templates from Hibernate

mine from the example code in the documentation which statements are relevant. Furthermore, these code snippets can be scattered in the documentation of numerous API commands. Hence, it is left to the programmer to find, read, and learn how to use the API commands. Another limitation of the Javadoc is that it targets only one API. There can be situations where there are templates using more than one API.

The technique described in this paper will display relevant code templates based on the current editing context. These templates are determined automatically by evaluating source code that has used the API commands. For example, a programmer may want to get a list of objects from the database. One way to achieve this is shown in Figure 1. Similar operations may have already been used in the same project or a different project. The query and parameters (lines 31 and 33) will need to be changed to fetch different objects, but the surrounding statements will remain the same for a similar operation. With the tool discussed in this paper, the programmer would be able to view the similar usages. Since the tool runs as a background process and does not expect any input from the programmer during its execution, the results are displayed with little interruption during the software development process. The templates are useful for the programmer because they save typing time and also allow the programmer to capture important coding rules[6].

3. N-GRAM LANGUAGE MODEL

Language models are a powerful statistical tool that can estimate the probability of sequences of words. These models have been successfully used in a variety of human language related problems such as augmentative communication for the disabled, text classification, hand-writing recognition, and spelling error detection [1, 5, 7].

We can assume that the probability of observing the current word is only dependent on the $n-1$ previous words, resulting in an N-gram language model (LM). The parameters in the N-gram language models are estimated from a training corpus using maximum likelihood estimation. We can evaluate the language model by measuring the perplexity (ppl) on a test sample.

Our approach follows the suggested general steps of applying a machine learning technique in software engineering as outlined by Zhang and Tsai [9]. The activities related to each of these steps are further described in the following subsections, but an overview is summarized below.

The training data in this case would be clones detected by a clone detection tool, Simscan[3]. Language models are created for every clone group. As the programmer is editing code, a background process obtains input from the unsaved portion of the source file that is being edited. This input constitutes the test data in which code templates will be suggested. The clone detection tool's role is to find similar sections of code from a separate program that has utilized the targeted API. The SRILM[4] toolkit, a tool for training and creating language models is used for the same.

Zhang and Tsai proposed seven steps to incorporate machine learning techniques [9]. Problem formulation attempts to conform a problem to the constructs of a machine learning technique. Problem representation translates the training data and knowledge to be learned into a specific representation. Data collection obtains the data necessary for the machine learning technique. Domain theory attempts to provide more understanding of the problem domain. Performing the learning process utilizes the selected machine learning technique on both the training and test data. Analyzing and evaluating learned knowledge looks at how the learning process has helped in solving the problem and what further improvements can be considered. Fielding the knowledge base utilizes the process of obtaining the knowledge within the context of a software development system.

3.1 Problem formulation

In this context, the problem is given an incomplete code segment, which is the unsaved portion of code that is being edited by the programmer. The goal is to find the closest match if one exists from a generated database. This process should be determined within the time frame of a user typing a few characters. It is difficult to identify the order of language models, as it is dependent on the training data. However, the trigram approach that was used in the experimentation and described in this paper was found to be effective. The incomplete code segment as input could be one or more identifiers or a complete statement from the eventual identified template that the programmer intends to use.

3.2 Problem representation

The input data consists of source files or blocks of code from a source file that represents the usage of templates. In this case, these code fragments come from source code that have used the API commands relevant to the current context in the programming session. The tool should learn from the code and should infer the templates for the user given the context. In other words, given a set of words, what is the likelihood of one of the templates being available? The similarity of one template to the given set of words does not necessarily qualify it as the intended template. There can be more than one template for a given set of words. For the Hibernate example in Figure 1, given the first two lines there can be three templates: delete, insert and list operations.

3.3 Data collection

In the ideal case, we need to group similar changes and use them to train the tool. But, performing such identification

[3]http://www.blue-edge.bg/simscan
[4]http://www.speech.sri.com/projects/srilm/

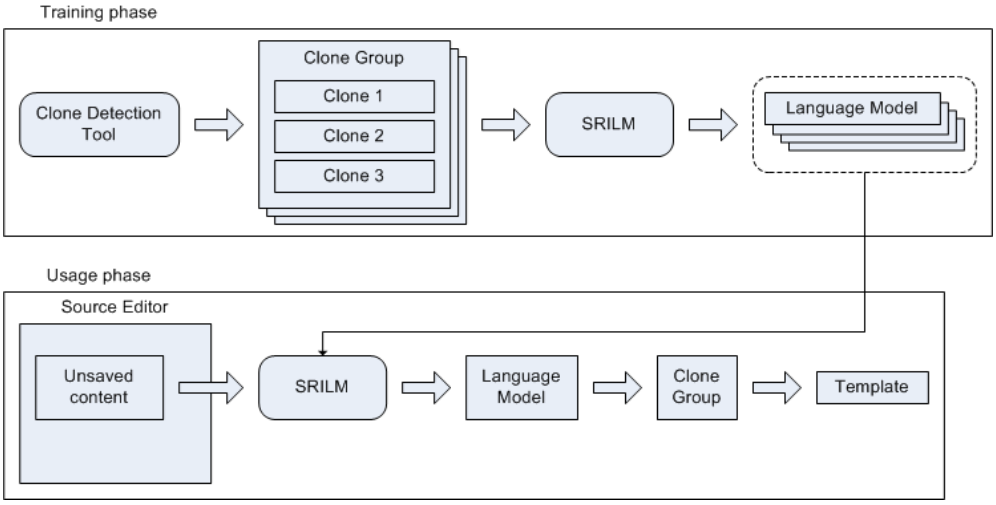

Figure 2: Overview of the tool support

manually in a given project would make the tool less useful. Code clone detection tools help to automatically find frequently used code in the project or folders containing source files. Tools like Simscan can find code clones in a more automated way. The utility of such tools in the problem described in this paper is evident as code templates represent code blocks that have been used in several locations in the source code. These code blocks can be considered code clones. However, the experiments that will be described later show that the prediction performance of the tool is directly dependent on the data (i.e., clone detection tool results) used for the training process.

The experiment and analysis in sections 3.5 and 3.6 are based on a study of 687 clones that were detected by Simscan in the JHotDraw[5] project. The distribution of the data is shown in Figure 3. The clones are divided into 195 clone groups, each of which represents a collection of clones of the same duplicated code. A clone group is a collection of clones that represent the same duplicated code. As seen in Figure 3, the number of clones in each clone group varies.

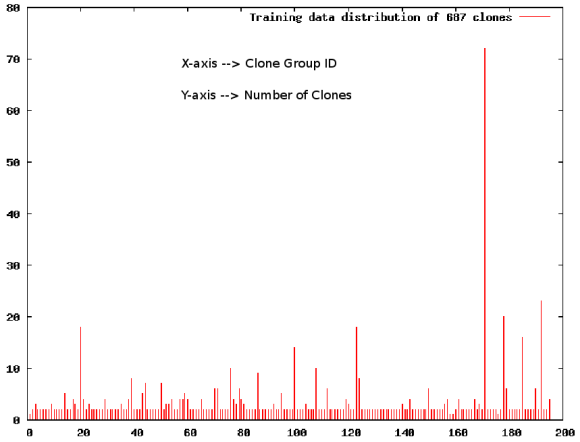

Figure 3: Clone distribution for JHotDraw given by Simscan

3.4 Domain preparation

SRILM by default is configured for natural languages. Some domain-specific changes were included from the default configuration for SRILM. An important change was related to the use of delimiters, as SRILM by default is configured with two delimiters (white space and end of sentence). The delimiters used for the analysis in this project were expanded to program language related delimiters, such as ".","(", ")", ";", and "=".

3.5 Performing the learning process

In the learning process, experiments were performed to determine the ppl value threshold that would serve as a good approximation for code template identification. The source code from different clones was used as training data and fed to SRILM. Tests were conducted with various hypothetical test data. The test data was selected such that it simulates the same effect as the incomplete code that the user is typing. To evaluate the tool, one random file representing a clone from each clone group was tested against all the clones. In the ideal condition it should return the clone group in which it is actually reported as a member in the clone detection tool results. For each file, the ppl value is calculated and the tool predicts the language model with the lowest ppl value (lower than a threshold value) as the inferred template. However, there are cases where the given language model or clone group was not the same as the actual clone group where the clone belongs. This may be affected by two factors:

- *Similar clone groups*: If the clone groups contain very similar clones with another group, the tool may match the clone with this other clone group.

[5]http://www.jhotdraw.org

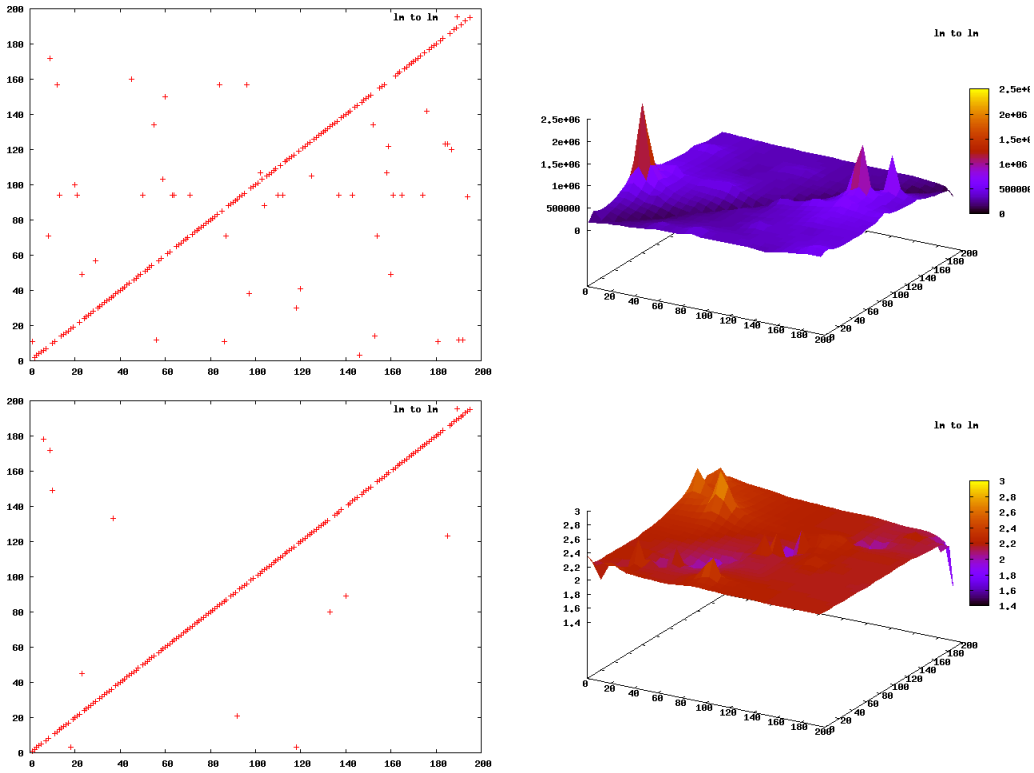

Figure 4: Refining the n-gram results

- *Lack of proper training data*: If there is not enough training data for a clone group, clones can be matched with other groups.

The results of the experiments are shown in Figure 4. Both the X and Y axes represent the clone group ID. In the ideal case, when any clone from a given group is tested, the tool should be able to identify the same clone group. Hence, in the ideal case the graph should be a diagonal line as every point on the X axis should be mapped to the same point on the Y axis. The left side of the figure shows the results without the ppl value, which is a representation of the difference from the language models. A lower value of ppl shows similarity, but for the best results that value should be small. A small experiment (conducted with a subset of the actual training data showed a ppl value of **2.8**) was a good approximation and it was later verified with the rest of the training data. The right side of the figure shows results with the ppl value; in the ideal case all the matching entries should have lower ppl values. The graph should be like a paper folded inwards diagonally.

3.6 Analysis and evaluation

Various n-gram configurations were evaluated to determine whether other n-gram approaches would yield better results. In Figure 4 (first row), a character-based trigram language model was used. The ppl values are high, and there are many mismatches. Another experiment was conducted for the same data with a trigram for every keyword and identifier (i.e., word-based trigram language model). The ppl values came down, but mismatches were still high. The 4-gram word approach was also tried, but also gave more mismatches, probably due to the need for a larger training set. Further evaluation of larger gram count values were not performed based on this observation. A noticeable improvement was found after including the domain-specific information explained in the last section and shown in Figure 4 (second row). Hence, this is used for further analysis. Based on the observations of the above experiments, we consider further enhancements to the n-gram approach that could potentially improve the current results.

- *Keywords and Identifiers*: The tool could potentially work better if a distinction can be achieved between keywords and identifiers in Java. This can be done by adding additional weight values for the identifiers.
- *Training Data Processing*: An additional process should be included before the training so that some of the irrelevant data can be removed.

3.7 Fielding the knowledge base

The tool is implemented as an Eclipse plugin project. The plugin communicates with the SRILM tool through a background process. Bash shell scripts[6] were written to find the relevant language model from the list of language models using the SRILM calculated ppl values.

[6]http://en.wikipedia.org/wiki/Bash

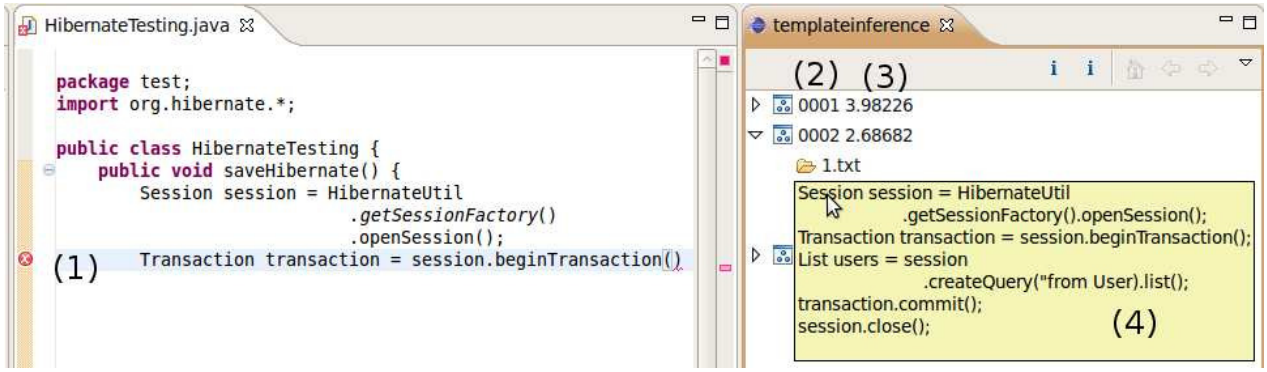

Figure 5: Template inference view in Eclipse IDE

The tool's output for the motivating example explained in Section 2 is shown in Figure 5. The training data is obtained from the Hibernate package of a chat program. The left side of the figure is the source editor where a developer types the code. The right side is the Eclipse view of the template inference tool. As shown in the figure, when the programmer types the last line of the code as shown in the editor, the view will show three template options. Of the three options, the only relevant option is the second one (as the ppl value should be < 2.8).

In the figure, '1' shows a compilation error due to the missing semicolon. This demonstrates that the tool can work even when the source code does not represent a well-formed AST. Every node in the template view consists of an id ('2') and the ppl value ('3') which is not surprising as in this case the difference is only a semi-colon. Hovering on top of one of these nodes will display the code usage in pop up box ('4')

4. EVALUATION

In this section, we evaluate the approach with three other current techniques that could be utilized to determine code templates. The evaluation of the supporting tool is based on the four criteria listed below:

- *AST*: This checks whether the tool considers the abstract syntax tree representation of the code. The AST information is important because it can support auto complete within the editor.
- *API-free*: This checks whether the tool is specific to a given API. For example, Javadoc is generated for one API, but cannot represent a functional block including more than one API.
- *Auto*: This checks whether the tool can work in the background and give a programmer necessary hints or help. The tool should not expect the programmer to leave the editor and search for lists of possible completions for a statement or a method.
- *Usage*: This option checks whether the tool distinguishes between a very frequently used code block with a least frequently used one.

Table 1: Comparitive study

Tool Name	AST	API-free	Auto	Usage
Javadoc				
Method Complete[2]	√	√		
Strathcona[3]				√
Template Support	√	√	√	√

The tools used for the study are Javadoc, Automatic Method Completion (AMC) [2] and Strathcona [3]. Javadoc's are usually generated from source code comments showing different methods and fields inside a class. The Strathcona and AMC are explained in Section 5. As shown in Table 1, our approach supports each of the criteria specified above. Javadoc is mainly a user guide and does not show much of the usage details. In the case of AMC and Strathcona, a programmer is expected to search with the incomplete code block. These tools could be useful for a programmer who is unfamiliar with the API usage, but what about a programmer who is aware of the API and only needs assistance for the purposes of saving time in typing? The tool support could be helpful in both cases for finding usages and also saving the typing time.

5. RELATED WORK

Hsu et al. [4] describe an approach to find repeating patterns in musical data using the n-gram based approach. A melody can be considered as a sequence of notes that sound pleasing when placed together. There are a number of ways a Melody can be repeated in a musical composition in a number of ways. It can be a repetition of the entire melody, which can be repeated at different intervals. Hsu et al. propose to encode all the notes and relevant information into text and input into an n-gram engine with the encoding. Our work is similar in that it also attempts to infer repeating patterns, but in this case the patterns are within the source code. Hence, the difference lies in the problem area and implementation details.

Hill and Rideout explain a strategy to include completion of the body of a method by using machine learning algorithms on the clones from large software projects. Instead of the n-gram approach, Hill and Rideout use the k-nearest neighbor algorithm in the fingerprint representation of the methods

[2]. The disadvantage of this approach is that because the approach makes use of vector representation for matching, it cannot find a match between a method and a block as the implementation defines them from different dimensions.

In [3], Holmes and Murphy describe an approach to provide relevant examples from a repository of source code to the programmer using the structure of the source code that the developer is writing. The approach extracts the context of the source code, which automatically frees the developer from learning a query language.

6. LIMITATIONS

Some of the limitations of the current implementation are explained below:

- *Reliability of Training Data*: The tool completely relies on the clone detectors to give reasonable training data. But there have been cases where clone detectors missed the code which should have been of interest to the tool. A few other cases were noticed where the code of interest were submerged in other code sections of the reported clones. Any of the above cases will result in the tool not finding the correct language model and template.

- *Unable to Generalize from the Clones*: It is not easy to generalize clones by determining its parameterized elements (e.g., identifier differences) and propose a template from the code associated with the clones. Some work to identify clone differences, such as described by Tairas [8] has been done, but not all code differences are supported. Hence in this case, even if we find the language model we may not be able to propose a template complete with its parameterized components.

7. CONCLUSION AND FUTURE WORK

This paper has successfully applied natural language processing techniques to the field of software engineering. The described tool assists the programmer by predicting the source code by using language models. Based on the experiments carried out, word-based trigrams with language specific delimiters provided the best prediction for code templates. In addition, a comparative study showed the added features and scope of the supporting tool.

Future work will consider adding an additional post-processing step after obtaining clones from clone detectors. This step will remove unwanted clones and also modify relevant clones so that they can serve as better training data.

8. ACKNOWLEDGMENTS

We thank Thamar Solorio and Jeff Gray for their suggestions and feedback for this project. The work described in this paper was supported in part by NSF CPA award #0702764.

9. REFERENCES

[1] C. Brockett, W. B. Dolan, and M. Gamon. Correcting ESL errors using phrasal SMT techniques. In *ACL '06: Proceedings of the 21st International Conference on Computational Linguistics and 44th Annual Meeting of the Association for Computational Linguistics*, pages 249–256, July 2006.

[2] R. Hill and J. Rideout. Automatic method completion. In *ASE '04: Proceedings of the 19th IEEE International Conference on Automated Software Engineering*, pages 228–235, 2004.

[3] R. Holmes and G. C. Murphy. Using structural context to recommend source code examples. In *ICSE '05: Proceedings of the 27th International Conference on Software Engineering*, pages 117–125. ACM, 2005.

[4] J.-L. Hsu, A. L. P. Chen, and C.-C. Liu. Efficient repeating pattern finding in music databases. In *CIKM '98: Proceedings of the seventh International Conference on Information and Knowledge Management*, pages 281–288, 1998.

[5] D. Jurafsky and J. H. Martin. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*. 2000.

[6] B. Livshits. Turning eclipse against itself: Improving the quality of eclipse plugins, 2005.

[7] C. D. Manning and H. Schütze. *Foundations of Statistical Natural Language Processing*. 1999.

[8] R. Tairas. Centralizing clone group representation and maintenance. In *OOPSLA '09: Companion to the 24th International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 781–782. ACM, 2009.

[9] D. Zhang and J. J. P. Tsai. Machine learning and software engineering. In *ICTAI '02: Proceedings of the 14th IEEE International Conference on Tools with Artificial Intelligence*, page 22, 2002.