CSCC: Simple, Efficient, Context Sensitive Code Completion

Muhammad Asaduzzaman Chanchal K. Roy Kevin A. Schneider Daqing Hou†
Department of Computer Science, University of Saskatchewan, Canada
†Electrical and Computer Engineering Department, Clarkson University, USA
{md.asad, chanchal.roy, kevin.schneider}@usask.ca, dhou@clarkson.edu

***Abstract*—Code Completion helps developers learn APIs and frees them from remembering every detail. In this paper, we describe a novel technique called CSCC (Context Sensitive Code Completion) for improving the performance of API method call completion. CSCC is context sensitive in that it uses new sources of information as the context of a target method call. CSCC indexes method calls in code examples by their contexts. To recommend completion proposals, CSCC ranks candidate methods by the similarities between their contexts and the context of the target call. Evaluation using a set of subject systems and five popular state-of-the-art techniques suggests that CSCC performs better than existing type or example-based code completion systems. We also investigate how the different contextual elements of the target call benefit CSCC.**

I. Introduction

Developers rely on frameworks and libraries of APIs to ease application development. While these APIs provide ready made solutions to complex problems, developers need to learn to use them effectively. The problem is that due to the large volume of APIs, it is practically impossible to learn and remember them completely. To avoid developers having to remember every detail, modern integrated development environments provide a feature called Code Completion, which displays a sorted list of completion proposals in a popup menu for a developer to navigate and select. In a study on the Eclipse IDE, Murphy et al. [1] find that code completion is one of the top ten commands used by developers, indicating that the feature is crucial for today's development. In this paper, we focus our attention to method call completion since it is the most frequently used form of code completion [11] (other forms of code completion include word completion, method parameter completion, and statement completion). In the remaining paper, we use the term Code Completion to refer to method call completion unless otherwise stated.

Existing code completion techniques can be divided into two broad categories. The first category uses mainly static type information, combined with various heuristics, to determine the target method call, but does not consider previous code examples or context of a method call. A popular example is the default code completion system available in Eclipse, which utilizes a static type system to recommend method calls. It sorts the completion proposals either alphabetically or by relevance before displaying them to users in a popup menu. Hou and Pletcher [13] develop another technique, called Better Code Completion (BCC), that uses a combination of sorting, filtering and grouping of APIs to improve the performance of the default type-based code completion system of Eclipse.

```
String line;
StringBuilder sb = new StringBuilder();
br= new BufferedReader(new
        FileReader("file.txt"));
try{
while ((line = br.readLine()) != null){
    sb . append ( l i n e ) ;
    sb . append ( '\n ' ) ;
}
}finally {br . close ();}
```

Fig. 1. An example of reading a file

The second category of techniques takes into account previous code examples and uses context matching to recommend target method calls. For example, to make recommendations, the Best Matching Neighbor (BMN) [12] code completion system matches the current code completion context to previous code examples using the k-Nearest Neighbour (kNN) algorithm.

BMN has successfully demonstrated that the performance of method call completion can be improved by utilizing the context of a target API method call. BMN focuses on using a special kind of context for a given call site, i.e., the list of methods that have been invoked on the same receiver variable plus the enclosing method of the call site. But there are many other possible forms of context to be considered. As an example of other forms of context, consider the code shown in Figure I, where a file is read via the `BufferedReader` object `br` in a `while` loop. In fact, the `readLine` method is commonly called as part of a `while` loop's condition located inside a try-catch block. Within a few lines of distance of the `readLine` method, developers usually create various objects related with that method call. For example, developers typically create a `BufferedReader` object from an instance of `FileReader` and later use that object to call the `readLine` method. Therefore, in addition to the methods that were previously called on the receiver object `br`, keywords (such as `while`, `try`, `new`), other methods (such as `FileReader`, `BufferedReader` constructor name) can be considered as part of the context of `readLine` as well. Adding these extra pieces of information can enrich the context of the targeted call to help recommend methods that are more relevant (`readLine`, in this case).

In this paper, we further explore the performance implications of these additional forms of context for code completion. To this end, we first propose a context sensitive code completion technique, called CSCC, that leverages code examples collected from repositories to extract method contexts to support code completion. Given a method call, we capture as its context *any method names, Java keywords, class or interface names* that appear within four lines of code. In this way, we build a database of context-method pairs as potential matching candidates. We use tokenization rather than parsing and advanced analysis to collect the context data, so our technique is simple. When completing code, given the receiver object, we use its type name and context to search for method calls whose contexts match with that of the receiver object. To scale up the search, we use *simhash* technique [28] to quickly eliminate the majority of non-matching candidates. This allows us to further refine the remaining much smaller set of matching candidates using more computationally expensive textual distance measures. This also makes our technique efficient and scalable. We sort the matching candidates using similarity scores, and recommend the top candidates to complete the method call.

We compare our technique with five other state-of-the-art techniques using eight open source software systems. Results from the evaluation suggest that our proposed technique performs better than any state-of-the-art static type or example-based systems that we have compared. Moreover, to understand how exactly the context of a method call affects code completion, we propose a taxonomy for the contextual elements of a method call and compare how different techniques perform for each category of contextual elements. This experiment helps us clearly understand the strengths and limitations of CSCC and other existing techniques.

This paper makes the following contributions:

1) A technique called CSCC to support code completion using a new kind of context and previous code examples as a knowledge-base.
2) A quantitative comparison of the proposed technique CSCC with five existing state-of-the-art tools that shows the effectiveness of our proposed technique.
3) A taxonomy of method call context elements and an experiment that helps to identify strengths and limitations of CSCC and other existing techniques.

The remainder of the paper is organized as follows. Section II briefly describes related work. Section III describes our proposed technique CSCC. Section IV compares CSCC with various code completion techniques. We discuss several key issues in our study in Section V. Section VI summarizes the threats to validity. Finally, Section VII concludes the paper.

II. Related Work

The study most relevant to ours is that of Bruch et al. [12]. They propose the Best Matching Neighbours (BMN) completion system that uses the k-nearest neighbour algorithm to recommend method calls for a particular receiver object. The most fundamental difference between BMN and CSCC lies in their definition of context. Our definition of a method call context includes any method names, keywords, class or interface names within the top four lines of a method call, whereas BMN's context is made of the set of methods that have been called on the receiver variable plus the enclosing method. Due to this difference, BMN and CSCC use different techniques to calculate similarities and distances. Lastly, BMN uses frequency of method calls to rank completion proposals, whereas CSCC ranks them based on distance measures.

Hou and Pletcher [3], [13] propose a code completion technique that uses a combination of sorting, filtering and grouping of APIs. They implement the technique in a research prototype called Better Code Completion (BCC). BCC can sort completion proposals based on the type-hierarchy or frequency count of method calls. It can filter non-API public methods. However, BCC does not leverage previous code examples. Moreover, BCC requires the filters to be manually specified which can only be performed by expert users of code libraries and frameworks. However, because CSCC considers the usage context of method calls to recommend completion proposals, methods that are not appropriate to call in a particular context would be automatically filtered out. So CSCC would require less effort to use than BCC.

Nguyen et al. [9], [10] use a graph-based algorithm to develop a context sensitive code completion technique, called GrePacc. The technique mines API usage patterns in open source code bases to create an API usage database. During the development phase, the technique extracts context sensitive features and matches them with usage patterns in the database. It then recommends a list of matched patterns to complete the remaining code. Although both GrePacc and CSCC utilize code context to make recommendations, the goals and approaches are different. GrePacc recommends multiple statements at a time, but CSCC completes a single method call.

Robbes and Lanza [11] propose a set of approaches to support code completion that use program history to recommend completion proposals. The program change history can be considered the *temporal* context for a method call, whereas ours is the *spatial* context. While their technique requires a change-based software repository to collect program history, our technique can work with any repository.

There are also a number of other techniques or tools that make use of previous code examples, but their goals are different than ours. For example, Mooty et al. [7] developed an Eclipse plugin, called Calcite, that helps developers to correctly instantiate a class or interface using existing code examples. While Calcite helps instantiate a class, we help developers complete method calls. Precise [5] mines existing code bases to recommend appropriate parameters for method calls. Hill and Rideout [4] focus on automatic completion of a method body by searching similar code fragments or code clones in a code-base.

Keyword programming [2] is also related to our study, but it defines a completely different way of user interaction for code completion. Instead of typing a method name, users type

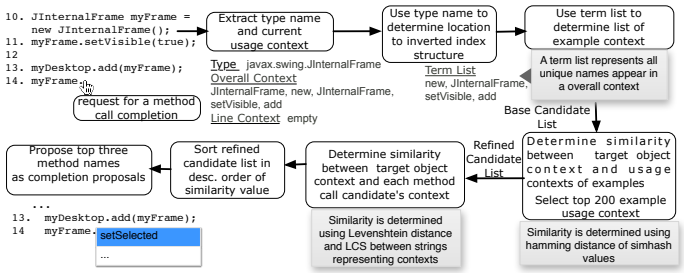

Fig. 2. Overview of CSCC's entire process of making recommendation starting from code completion request.

some keywords that give hints about the method the user is trying to call. The algorithm then automatically completes the method call or makes appropriate suggestions to complete the remaining part. Han and Miller [8] later introduce abbreviation completion that uses a non-predefined set of inputs to complete the target method call.

III. Proposed Algorithm

In this section, we describe our algorithm for finding method calls to recommend for a target object. Figure 2 presents an overview of the process. Our example-based, context-sensitive code completion system works in three steps:

- Collect the usage context of API method calls from code examples and index them by their contexts to support quick access. We model the context of an API method call by method calls, Java keywords and type names that appear within the four lines prior to the receiver object that called the method. We hypothesize that these elements around the target method call can provide a better, fuller context than other approaches.
- Search for method calls whose context matches with that of the target object. One approach would be to directly measure the similarity between the context of the target object and that of each method call in the example code base using string edit-distances. However, string edit-distance operations are computationally expensive. To speed up the search, we instead use the Hamming distance over the *simhash* values as similarity measures. We determine a smaller list of method names that are the more likely candidates for code completion, which we refer to as the *candidate list*.
- The final step synthesizes the method calls from the candidate list. For each method name in the candidate list, we use a combination of token-based Longest Common Subsequence (LCS) and Levenshtein distance to determine a similarity value of its context with that of the receiver object. We then sort the method names in descending order of similarity value and recommend the top three names to complete the method call.

We describe the three steps in detail as follows.

A. Collect usage context of method calls

In this step, CSCC mines code examples to find the usage context of API method calls. To capture a method call context,

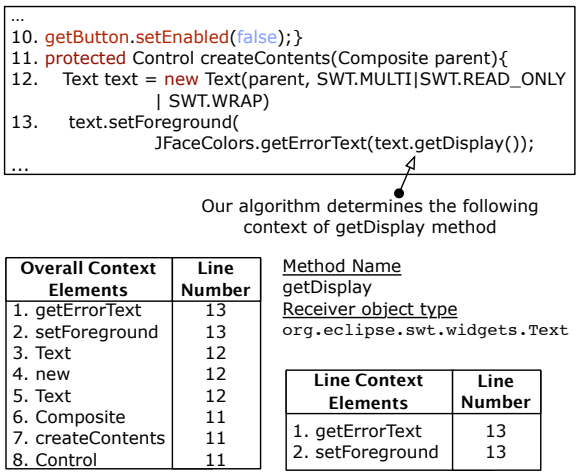

Fig. 3. Overall context and line context for *getDisplay*.

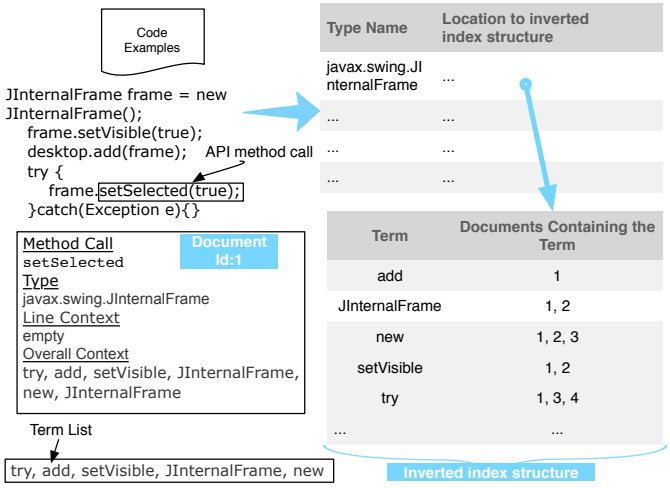

Fig. 4. Database of method call contexts are grouped by receiver types using inverted index structure.

we consider the content of the n lines prior to it, including the line where the target method call appears. In this study, we use $n = 4$ and we validate this decision in Section V.

We collect the following three kinds of information from the four lines of context, which we refer to as the *overall context* of the method call:

1) Any method names.
2) Any Java keywords except access specifiers.
3) Any class or interface names.

When extracting the overall context, we ignore blank lines, comment lines, or lines containing only curly braces. We also remove any duplicate tokens from the overall context.

In addition, we separately collect a *line context* for the target method, which includes any method names, keywords (except access specifiers), class or interface names and assignment operators that appear on the same line but before the target method call. When the overall contexts are completely different and fail to match, line contexts act as a secondary criterion for matching.

To further explain the construction of both overall and line context, consider the method call at line number 13 as shown in Figure 3. The contents of both contexts include tokens and their locations. Note that although line number 10 is within four lines of our target method call *getDisplay*, it is not considered part of the context as it is located outside of the *createContents* method containing the target call *getDisplay*.

We use a two-level indexing scheme to organize the collected usage contexts of method calls (see Figure 4 for an example of it). We use the type name of a receiver object to group all method calls that have been invoked on the type. We use an inverted index structure [21] to organize such a group of method calls. More specifically, an inverted index is a data structure that maps each term to its location in a document. We represent each overall context of a method call as a document, and use tokens from the context to index the set of documents where they appear.

B. Determine candidates for code completion

When a user requests a method call completion (for example, in Eclipse typing a dot (.) after an object name initiates such a request), our algorithm first extracts both overall and line contexts for the receiver object. To find candidate methods for code completion, we match the current context to those extracted from the example code base. Specifically, we use the type name of the receiver object as an index to determine the related inverted index structure, which contains all method calls made on the receiver type (Figure 4). We then use tokens from the overall context as keys to the inverted index structure to collect all those method calls in the code examples that have the same type as the receiver object. We refer to these matching method calls as the *base candidate list*.

The *base candidate list* often contains thousands of method calls, so we need to reduce them to a small number of most likely candidates in order to recommend. We follow a two-step process to search for the most likely candidates. We first use the *simhash* technique to determine a shorter list of method names (currently the top 200 that are deemed most similar to the target context) that are more likely candidates to complete the current method call and quickly eliminate the majority of others. To calculate string similarity metrics, we concatenate all the tokens of each context and generate a *simhash* value for the concatenated string (see Figure 5 for an example). We use the *simhash* technique to eliminate most of the irrelevant matching candidates because it is both fast and scalable. Second, we use the normalized Longest Common Subsequence (LCS) and Levenshtein distances to measure the fine-grained similarity between the target context and the context of each likely matching candidate to obtain a refined candidate list. Calculating LCS and Levenshtein distances are both time consuming operations. Although they provide fine-grained similarity measures, due to the near real-time constraint on code completion, we cannot apply them directly on the *base candidate list*.

We use the *simhash* technique [28] to identify the most likely method call candidates. *Simhash* uses a cryptographic

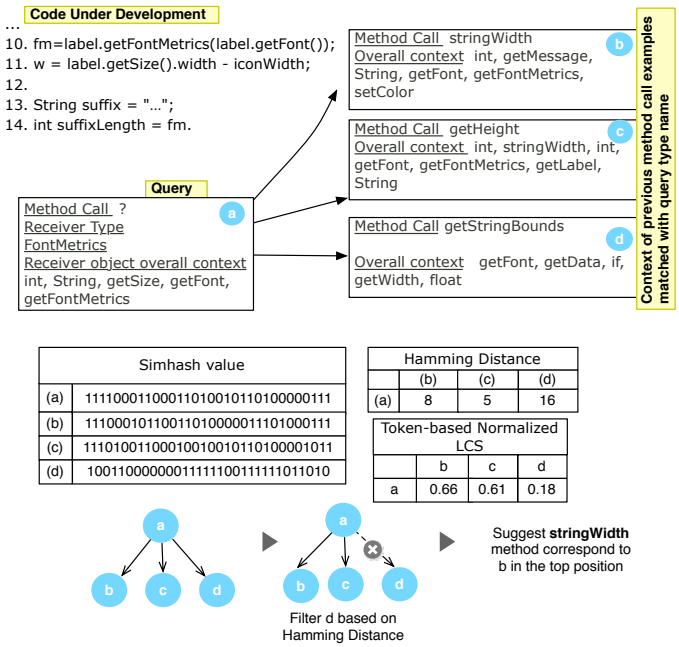

Fig. 5. Context similarities are measured using Hamming distance between simhash values.

hash function to generate binary hash keys, also known as *simhash* values. An important property of simhash is that strings that are similar to each other have either identical or very similar *simhash* values. Therefore, we determine the similarity between each pair of contexts using the Hamming distance of their corresponding *simhash* values. We use the Hamming distance of the overall context to sort the matching candidates unless the Hamming distance of the line context exceeds a predefined threshold value, in which case we use the Hamming distance of the line context as the distance measure. After sorting by similarity, we take the top k method contexts as the likely matching candidates of the target context. After experimentation with different values of k, we found that $k = 200$ is a good choice to work with and we use that value in our study. We refer to this list as the *refined candidate list*.

To recommend method calls, we further sort the method names in the refined candidate list by combining both overall and line context similarities as follows. We use the normalized Longest Common Subsequence (LCS) distance to measure the similarity of the token sequences from the overall context. We use Levenshtein distance to measure the similarity of the token sequences from the line context. We sort matching candidates in descending order of their overall context similarity. However, in case of a tie for the overall context similarity, we use the line context similarity. We ignore all matching candidates whose similarity value drop to a certain threshold. We empirically found that 0.30 is a good choice to work with.

The *simhash* technique has been found effective for detecting similar pages in a large collection of web documents [27] and also has been used successfully in detecting similar code fragments in code clone detection [26]. Although various hash functions are available, we use the *Jenkin* hash function since it

has been found effective in a previous study [26]. We generate a 64 bit *simhash* value for both overall and line contexts of the target object. To save computation time, we precompute the *simhash* values. We determine the similarity between each pair of contexts using the Hamming distance of the corresponding *simhash* values.

C. Recommend top-3 method calls

The objective of this step is to recommend a list of completion proposals (i.e., method names). Since there are many code examples associated with the same method call, the sorted list of method names obtained from the previous step may contain many duplicates. After eliminating duplicates, we present the top three method names to the users.

IV. Evaluation

We evaluate our technique and compare CSCC with five state-of-the-art code completion systems using eight open-source systems (Table I). For a given subject system, we determine all locations where methods from a target API have been called. This set of method calls constitutes our data set. We then apply the ten-fold cross validation technique to measure the performance of each algorithm. This is a popular way of measuring performance of information retrieval systems [14] and has been used previously in many research projects. First, we divide the entire data set into ten different folds, each containing an equal number of method calls. Next, for each fold, we use code examples from the nine other folds to train the technique for method call completion. The remaining fold is used to test the performance of the technique.

We use precision, recall, and F-measure to measure the performance of an algorithm. If a target method call is in the top-three completion proposals made by a completion system, we consider the recommendation relevant. The precision, recall, and F-measure are defined as follows:

$$Precision = \frac{recommendations\ made \cap relevant}{recommendations\ made} \quad (1)$$

$$Recall = \frac{recommendations\ made}{recommendations\ requested} \quad (2)$$

$$F\text{-}measure = \frac{2 \cdot Precision \cdot Recall}{Precision + Recall} \quad (3)$$

where $recommendations\ requested$ is the number of method calls in our test data for which we will make a code completion request. $Recommendations\ made$ is the number of times where a code completion system makes a recommendation.

A. Test Systems

We chose to focus on two API's, SWT and Swing/AWT, as the target for our evaluation. These are popular libraries extensively used for developping GUI applications. We selected four systems that used SWT. The largest one is Eclipse 3.5.2 [19], a popular open source IDE. Vuze [16] is a P2P file sharing client using the bittorrent protocol. Subversive [17] provides support to work with Subversion directly from Eclipse. RSSOwl [18] is an RSS newsreader.

We also chose four open source software systems for AWT/Swing. NetBeans 7.3.1 [20], the largest, is an IDE; jEdit [22] is a text editor; ArgoUML [23] is a UML modeling tool; and JFreeChart [24] is a Java charting library.

B. Evaluation Results

In this section, we discuss the results of evaluating and comparing CSCC with five other code completion systems (ECCAlpha, ECCRelevance, FCC (Frequency-based Code Completion), BCC, and BMN) using the eight test systems. We have introduced BCC and BMN earlier. ECCAlpha and ECCRelevance are two default Eclipse code completion systems that leverage the static type system. ECCAlpha sorts the completion proposals in alphabetical order, and ECCRelevance uses a positive integer value, called relevance, to sort them. The value is calculated based on the expected type of the expression as well as the types in the code context (such as return types, cast types, variable types etc.). The Frequency-based code completion system (FCC) considers the frequency of method calls in a model to make recommendations. The more frequent a method occurs, the higher its position is in the completion proposals.

Table I shows the precision, recall, and F-measure values for the six code completion systems. The top four rows are results collected for SWT, and the bottom four rows for AWT/Swing. Overall, CSCC achieves higher precision and recall values than any of the other techniques for both single and top three completion proposals. For the top three proposals, it has precision of 73-86% and recall of 97-99%. The recalls for ECCAlpha, ECCRelevance and BCC are all one. But both ECCAlpha and ECCRelevance performed poorly, and BCC outperformed both of them. Except in a few cases, BCC also outperforms FCC. Furthermore, the performance of FCC is not great, while its recall is close to 100%, its precision is only 49-62% for the top three proposals. Interestingly, BMN did not perform well either. Although its precision is better than both BCC and FCC for the single and top-3 suggestions, its recall is poorer in both cases. This is due to the fact that BMN targets local variable method calls but there are many places in source code where methods are called on fields, parameters, chained expressions, or even static types. We performed further experiments to elaborate on this issue in Section IV-C1.

The results for AWT/Swing shown in the bottom four rows of Table I are consistent with those of SWT. For example, for the top three proposals and for the largest subject system (NetBeans), the F-measure of CSCC is higher by 15% compared to the closest performing technique.

To test whether CSCC performed significantly better than other techniques, we also performed directional Wilcoxon Signed Rank Tests for the top three completion proposals. CSCC is significantly better than all other techniques evaluated. For example, CSCC performs statistically significantly better than BMN in terms of both precision and recall values (W=13, N=8, α=0.05 in both cases).

TABLE I
EVALUATION RESULTS OF CODE COMPLETION SYSTEMS. *Delta* SHOWS THE IMPROVEMENT OF CSCC OVER BMN.

Subject Systems		Precision							Recall							F-Measure						
		ECCAlpha	ECCRel	BCC	FCC	BMN	CSCC	Delta (%)	ECCAlpha	ECCRel	BCC	FCC	BMN	CSCC	Delta (%)	ECCAlpha	ECCRel	BCC	FCC	BMN	CSCC	Delta (%)
Eclipse	Top-1	0.006	0.01	0.24	0.31	0.36	0.60	24	1	1	1	1	0.77	0.99	22	0.008	0.020	0.39	0.47	0.49	0.75	26
	Top-3	0.052	0.20	0.52	0.49	0.63	0.80	17	1	1	1	1	0.77	0.99	22	0.10	0.34	0.68	0.66	0.69	0.88	19
	Top-10	0.14	0.34	0.80	0.73	0.69	0.90	21	1	1	1	1	0.77	0.99	22	0.25	0.51	0.89	0.84	0.73	0.94	21
Vuze	Top-1	0.005	0.10	0.23	0.33	0.35	0.56	21	1	1	1	1	0.76	0.98	22	0.009	0.18	0.37	0.50	0.48	0.71	23
	Top-3	0.03	0.16	0.49	0.49	0.59	0.73	14	1	1	1	1	0.76	0.98	22	0.06	0.28	0.66	0.66	0.66	0.84	18
	Top-10	0.20	0.36	0.94	0.71	0.61	0.83	22	1	1	1	1	0.76	0.98	22	0.33	0.53	0.97	0.83	0.68	0.90	22
Subversive	Top-1	0.01	0.03	0.30	0.36	0.58	0.68	10	1	1	1	1	0.38	0.97	60	0.02	0.058	0.46	0.53	0.46	0.80	34
	Top-3	0.02	0.07	0.63	0.62	0.77	0.86	9	1	1	1	1	0.38	0.97	60	0.04	0.13	0.77	0.77	0.51	0.91	40
	Top-10	0.07	0.20	0.96	0.88	0.79	0.91	12	1	1	1	1	0.38	0.97	60	0.13	0.34	0.98	0.94	0.51	0.94	43
Rsowl	Top-1	0.01	0.078	0.25	0.32	0.48	0.65	17	1	1	1	1	0.72	0.98	26	0.20	0.14	0.40	0.48	0.58	0.78	20
	Top-3	0.024	0.16	0.58	0.51	0.74	0.84	10	1	1	1	1	0.72	0.98	26	0.046	0.28	0.73	0.68	0.73	0.90	17
	Top-10	0.077	0.29	0.85	0.74	0.80	0.90	10	1	1	1	1	0.72	0.98	26	0.14	0.45	0.92	0.85	0.76	0.94	18
NetBeans	Top-1	0.12	0.13	0.34	0.29	0.43	0.66	23	1	1	1	1	0.67	0.98	31	0.21	0.23	0.51	0.45	0.52	0.79	27
	Top-3	0.18	0.25	0.62	0.53	0.67	0.86	19	1	1	1	1	0.67	0.98	31	0.31	0.40	0.77	0.69	0.67	0.92	25
	Top-10	0.36	0.48	0.86	0.73	0.70	0.92	22	1	1	1	1	0.67	0.98	31	0.70	0.65	0.92	0.84	0.68	0.95	27
JEdit	Top-1	0.009	0.12	0.41	0.35	0.52	0.62	10	1	1	1	0.98	0.70	0.94	24	0.02	0.21	0.58	0.52	0.60	0.75	15
	Top-3	0.14	0.29	0.62	0.53	0.74	0.79	5	1	1	1	0.98	0.70	0.94	24	0.25	0.45	0.77	0.69	0.72	0.86	14
	Top-10	0.32	0.49	0.83	0.74	0.79	0.85	6	1	1	1	0.98	0.70	0.94	24	0.48	0.66	0.91	0.84	0.74	0.89	15
ArgoUML	Top-1	0.03	0.13	0.40	0.32	0.46	0.58	12	1	1	1	0.99	0.68	0.95	27	0.058	0.23	0.57	0.48	0.55	0.72	17
	Top-3	0.12	0.27	0.65	0.53	0.68	0.74	6	1	1	1	0.99	0.68	0.95	27	0.21	0.43	0.79	0.69	0.68	0.83	15
	Top-10	0.27	0.47	0.83	0.78	0.74	0.81	7	1	1	1	0.99	0.68	0.95	27	0.43	0.64	0.91	0.87	0.71	0.87	16
JFreeChart	Top-1	0.02	0.08	0.35	0.32	0.42	0.63	21	1	1	1	1	0.75	0.98	23	0.040	0.15	0.52	0.48	0.54	0.77	23
	Top-3	0.05	0.19	0.52	0.63	0.76	0.85	9	1	1	1	1	0.75	0.98	23	0.10	0.32	0.68	0.77	0.75	0.91	16
	Top-10	0.27	0.58	0.63	0.92	0.84	0.94	10	1	1	1	1	0.75	0.98	23	0.43	0.73	0.77	0.96	0.79	0.96	17

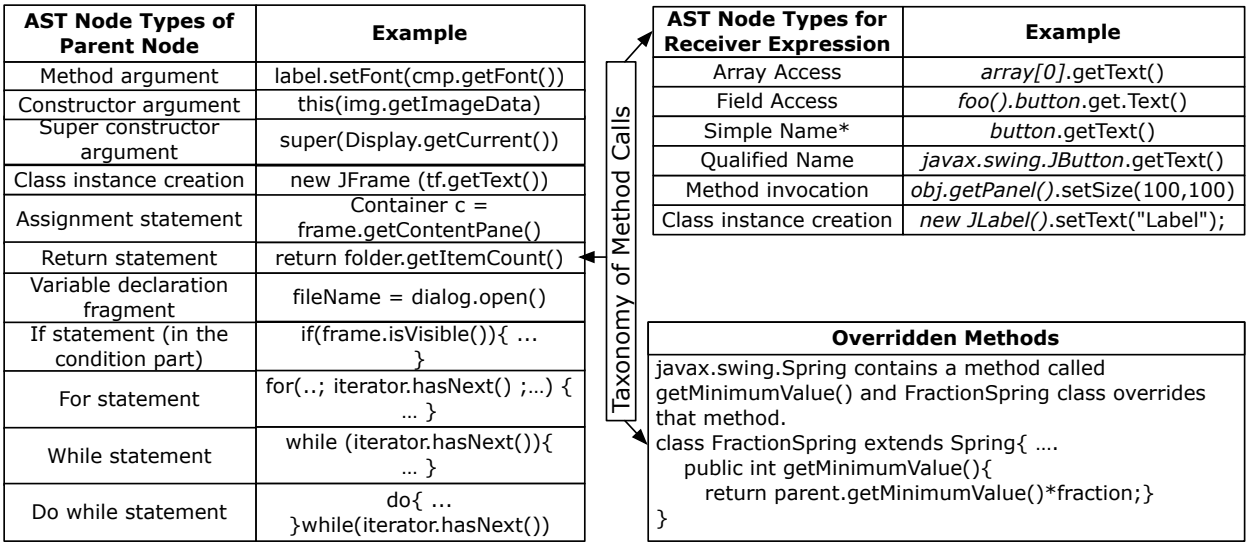

Fig. 6. Taxonomy of method calls

C. Evaluation Using a Taxonomy of Method Calls

While the evaluation in Section IV-B provides a ranking of the six techniques in terms of their performance, it does not reveal what factors contribute to CSCC's better performance. We hypothesize that it is due to CSCC's ability to capture a fuller context for method calls. To further shed light on this hypothesis, we propose a taxonomy for the characteristics of a method context, and compare the techniques using each category of method call characteristics within the taxonomy.

Our taxonomy (Figure 6) includes three categories of characteristics for the context of a target method call: the AST node types for its receiver expression, the AST node types for its parent node, and the enclosing overridden method that contains the target method call. Although we cannot guarantee that the taxonomy covers every possible aspect of method call completions, it can provide insights into code completion techniques and can also help us to decide where more effort is needed.

We use the following procedure for our evaluation. For each category of method calls within each test fold, we count

TABLE II
CATEGORIZATION OF METHOD CALLS FOR DIFFERENT AST NODE TYPES FOR RECEIVER EXPRESSIONS (FOR THE TOP-3 PROPOSALS)

Expression Types		NetBeans			Eclipse		
		Quantity (%)	Correctly Predicted BMN (%)	Correctly Predicted CSCC (%)	Quantity (%)	Correctly Predicted BMN (%)	Correctly Predicted CSCC (%)
Array Access		0.54	0	54.16	1.65	0	79.73
Class Instance Creation		0.18	0	100	0.11	0	100
Field Access		0.60	0	80.77	0.49	0	77.27
Method Invocation		21.66	0	94	11.88	0	75.42
Simple Name	Type	5.48	79.21	92.13	3.02	96.26	98.26
	Local Variable	32.13	68.26	84.85	41.86	0.64	83.14
	Field	51.94	52.07	79.95	46.92	50	75.46
	Parameter	10.44	52.80	79.35	9.47	46.81	79.50
	Total	74.08	58.84	82.13	85.02	56.86	79.65
Qualified Name		2.94	57.36	82.17	0.85	60.53	71.05

how many of them are correctly predicted by code completion techniques. We present the final result after summing up the results for all ten test folds.

1) AST Node Type for Receiver Expression: We categorize the receiver expressions of the test method calls according to their AST node types. We count the number of test method calls that each code completion technique correctly recommends for each kind of AST node. Table II shows the results for the top three proposals from the two largest test systems, Eclipse and NetBeans.

Table II suggests that the majority of receiver expressions fall in the simple name category. A simple name can be a variable name (declared as a method parameter, a local variable, or a field) or a type name (static method calls). The original BMN technique considers only local variables and their types to compute completion proposals. However, Table II shows that many receiver expressions of method calls are not local variables, and thus for which BMN produces no recommendations. This explains why we did not receive good results for BMN (Table I). The way BMN collects usage context is quite limited. In contrast, CSCC can identify usage context even when the receiver is not a local variable and thus can recommend method names for those cases too.

We performed another experiment where we train and test both BMN and CSCC using only those method calls where the receiver is a local variable. For the top three proposals, BMN achieves 68% recall and 77% precision for the Eclipse system, both of which are higher than those of any other techniques except CSCC. The recall and precision for CSCC are 84% and 86%, respectively, indicating that CSCC performs better than BMN even when the receivers are local variables.

TABLE III
CORRECTLY PREDICTED METHOD CALLS WHERE THE PARENT EXPRESSION EXPECTS A PARTICULAR TYPE OF VALUE (FOR THE TOP-3 PROPOSALS AND FOR THE ECLIPSE SYSTEM)

AST Node Types of Parent Node	Total Cases	ECCAlpha	ECCRel	BCC	FCC	BMN	CSCC
Method Argument	696	6	441	527	378	336	553
Assignment	429	3	282	338	144	184	305
If Statement	526	36	47	225	217	214	373
While Statement	8	0	0	4	0	2	4
Return	99	1	56	62	38	32	65
Variable Declaration Fragment	1388	10	738	1018	498	515	1084
For Statement	5	0	3	3	0	0	4
Class Instance Creation	126	3	76	95	46	54	98
Prefix Expression	425	30	75	396	282	228	346
Total	3702	89 2.4%	1718 46.4%	2668 72%	1603 44.5%	1565 42.27%	2832 76.5%

2) AST Node Types of Parent Node: We consider those method call expressions where a particular type of object or value is expected. For example, a framework method call can be located in the condition part of an if statement that expects a boolean value. A method call can be located in the right hand side of an assignment expression. If the left hand side of that assignment expression is of Container type, the right hand side should return an object of type Container or a sub-type of it. The goal is to identify how well techniques that consider type information as context perform in these cases compared to others. To make the result comparable with the BMN code completion system, we consider those method calls where the receiver is a local variable.

Table III shows the results of top three proposals for the Eclipse system. We can see that considering the expected type as contextual information can help improve code completion techniques. That is why the accuracy of BCC becomes close to that of CSCC, which achieves the highest accuracy for all but one category of AST node. Other code completion systems, such as BMN, FCC and default code completion systems of Eclipse, did not perform well in this experiment.

3) Method Calls inside Overridden Methods: The objective is to verify whether target methods called from within overridden methods impose any challenge to the code completion techniques. Our informal observation is that it may be difficult to identify usage context for method calls within overridden methods. Similar to the previous experiment, we again test only those method calls where the receiver is a local variable. Although CSCC performs better than any other techniques on previous experiments, it performs poorly in this experiment. When we manually analyze some of the code examples, we notice that method calls within overridden methods contain very limited contextual information. For example, a large number of methods in Java Swing applications result from

TABLE IV
PERCENTAGE OF CORRECTLY PREDICTED METHOD CALLS THAT ARE CALLED IN THE OVERRIDDEN METHODS (FOR THE TOP-3 PROPOSALS)

Subject Systems	Percentage of correctly predicted method calls by Code Completion Systems (%)					
	ECCAlpha	ECCRel	BCC	FCC	BMN	CSCC
Eclipse	6.15	16.73	47.40	24.62	23.56	17.60
NetBeans	5.60	25.06	52.80	17.45	15.44	19.69

TABLE V
RUNTIME PERFORMANCE OF CSCC GENERATING A DATABASE OF 40,863 METHOD CALLS (COLUMN 2) AND PERFORMING 4,540 CODE COMPLETION (COLUMN 3) FOR THE ECLIPSE SYSTEM.

Setup Used	**Database Generation Time**	**Code Completion Time**
CSCC (with inverted index)	**8,066 ms**	**8,998 ms (Avg.1.94 ms)**
Without inverted index	8,000 ms	12,888 ms (Avg.2.77 ms)

implementing the ActionListener interface and those methods contain only a few method calls or method calls that are not related with each other, which possibly contributed to such poor results and thus requires further investigation. However, static type based system BCC in such case performs better than other techniques. For example, BCC can correctly recommend method calls in 47.40% for Eclipse (52.80% for NetBeans) of the total methods that are called in overridden methods.

D. Comparison With Code Recommenders

Code Recommenders is a more advanced Eclipse plugin evolved from BMN. It utilizes a model trained using a set of code examples to provide intelligent code completion. When a developer invokes code completion in the IDE, Code Recommenders collects the current usage context and then looks in its model for possible matches to complete the code. Because the model is proprietary, we cannot train it with new code examples. Furthermore, because we did not have access to its internal API to obtain completion proposals automatically, we could only perform a manual comparison instead. Our comparison indicates that CSCC performed better than Code Recommenders. Our comparison is limited in scale due to its manual nature. Extensive evaluation may be possible in future if Code Recommenders becomes more open.

From the code examples in a book on the Java Swing framework [25], we randomly selected 309 Swing/AWT method calls as our test cases. We enabled the Code Recommenders intelligent call completion in an Eclipse IDE. Then for each test case, we manually opened the corresponding file in the IDE, removed any code after the target object and tried to complete the method call by typing a dot (.) after the object name. We recorded the list of completion proposals suggested by Code Recommenders and determined the rank of the target method name in that list. To obtain the performance result for CSCC, we trained CSCC with the remaining examples and tested CSCC against the 309 selected method calls. CSCC achieves better result than Code Recommenders. The precison, recall and F-measure for Code Recommenders are 62%, 76% and 68%, and for CSCC 92%, 82% and 87% respectively.

E. Runtime Performance

To be useful, code completion must be done at near real-time in order to not interrupt a developer's flow of coding. Thus, to study the time required to suggest completion proposals, we measured the runtime of the first and last two steps of CSCC. The first step is responsible for building a candidate method call database and the last two steps are about recommending completion proposals. All experiments were performed on a computer running Ubuntu Linux with a 3.40 GHz Intel Core i7 processor and 10 GB of memory.

As shown in Table V, we provide runtime data for the Eclipse subject system where the model is built using 40,863 method calls (column two) and the running time is the time required to test all 4,540 queries (column three). As expected, the first step takes the most time but the database needs to be built only once. On average, it takes 1.94 ms (milliseconds) to compute the completion proposals for each method call, which is negligible.

To understand the benefits of using the inverted index structure, we also developed a variant of our algorithm without using the inverted index structure and measured the runtime again. The result is summarized in the second row of Table V. While the model generation time reduces slightly, the code completion running time increases considerably. Using the inverted index structure not only reduces the runtime of the algorithm, but also improves the result slightly by eliminating many irrelevant mapping candidates.

V. DISCUSSION

A. Why does CSCC consider four lines as context?

For an API method call, we use four lines prior to the method call to determine the overall context. The number four is determined experimentally as follows. For this experiment, Eclipse is used as a subject system. We collect all SWT method calls and randomly select 10% for testing. The remaining 90% of calls are used to train CSCC. Next, we run the algorithm 10 times by varying context line numbers from one to ten. The higher the context line number, the larger the context size, which results in an increase in computation time. We need to keep the number of context lines as low as possible without impacting performance. Figure 7 shows the accuracy of CSCC at various context line numbers. From Figure 7, we can see that at the beginning, the number of correctly predicted method calls drops when we increase the context size from one to two lines. However, we observe a sharp increase from that point for increasing the context size. When the context size increases to more than four lines there is no significant change in the number of correct predictions. Therefore, we set the context size to four lines.

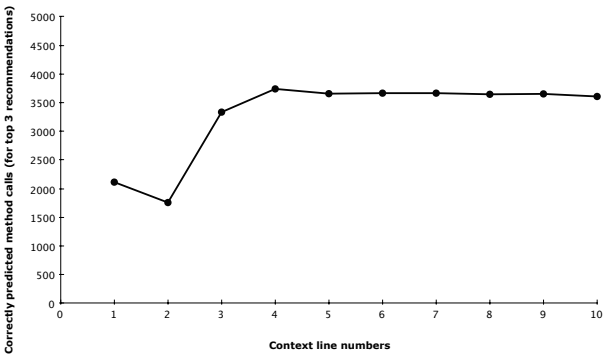

Fig. 7. The number of correct predictions at different context size

B. What is the impact of different context information?

We evaluate the impact of CSCC's various context information on the performance of method call prediction. We run an experiment on NetBeans, our largest Swing/AWT system.

Table VI shows the percentage of correctly predicted method calls for different combinations of context information. In the first row, we model the overall context considering those methods that were previously called on the receiver object. but without the enclosing method name. Next, we consider a variation of the previous model that takes into account enclosing method name. For the above two models we neither consider any line context nor put any limit on context size. But all the models in the following five rows use overall context to recommend completion proposals. The third row corresponds to a model that only considers those method names that were previously called within four lines of distance on the same receiver object of the target method call. The fourth row represents a model that in addition to the above information, also considers the line context. The fifth row corresponds to another model that in addition to the previous information, also considers any other method names located within four lines of distance. The model in the sixth row takes into account any type names (class or interface names) appearing as part of class instance creation plus the previous information. Finally, the last row implements the complete CSCC, which also includes any Java keywords except access specifiers.

According to the results, CSCC in the last row achieves the highest accuracy. It is also clear from the table that the performance of CSCC is increasing with the addition of additional context information. Since adding the enclosing method name does not improve performance significantly (compare the first two rows), we did not include enclosing method name in CSCC. Among various additional information we considered, method names and the type names (appears in the class instance expression) contributed the most. Although the addition of the line context improves the overall performance by only around 1%, during our manual investigation we found that in those small number of cases, overall context differs considerably, so line context complements the overall context in this case. Surprisingly, adding keyword names did not improve the performance significantly. We analyzed some cases manually and found that while they are effective, their effect diminishes in the matching process because of the presence of a large number of methods, and class/interface names in the context.

TABLE VI
SENSITIVITY OF PERFORMANCE TO DIFFERENT CONTEXT INFORMATION

Model	Correctly predicted method calls(%)			
	Top-1	Top-3	Top-5	Top-10
Rec. method calls	46.5	69.5	77.5	82.5
Rec. method calls + enclosing method	46.6	68.7	76.9	81.8
Rec. method calls (within four lines)	33	60.20	76	84.80
Rec. method calls (within four lines)+line context	34.8	61.62	76.26	84.89
Previous factors + Other method calls	58	78	83	87
Previous factors + Type name	62	82	86	89
Previous factors + keyword (CSCC)	64	84	88	90

TABLE VII
CROSS-PROJECT PREDICTION RESULTS. P, R, AND F REFER TO PRECISION, RECALL, AND F-MEASURE, RESPECTIVELY.

Subject Systems			FCC (%)	BMN (%)	CSCC (%)
NetBeans	Top-1	P	39	46	69
		R	100	90	98
		F	56	61	**81**
	Top-3	P	64	77	84
		R	100	90	99
		F	78	83	**91**
JEdit	Top-1	P	57	70	66
		R	100	84	98
		F	73	76	**79**
	Top-3	P	69	85	83
		R	100	84	98
		F	82	84	**90**
ArgoUML	Top-1	P	48	48	54
		R	100	85	100
		F	65	61	**70**
	Top-3	P	65	68	77
		R	100	85	100
		F	79	76	**87**
JFreeChart	Top-1	P	43	44	68
		R	100	89	100
		F	60	59	**81**
	Top-3	P	74	85	88
		R	100	89	100
		F	85	87	**94**

C. How effective is the technique in cross-project prediction?

We performed another experiment to evaluate the effectiveness of CSCC in cross project prediction. We followed the approach described by Nguyen et al. [29] by using ten-fold cross validation. For each fold, we trained with nine other folds of the same system plus code examples from all other systems. To make the results comparable with BMN, we only provide prediction results for local variable method calls. Table VII shows the results of our method call prediction.

CSCC once again performs better than other techniques. There are two important lessons to be learned from the result. First, both precision and recall of CSCC either slightly increase or are consistent with those of Table I, indicating

that CSCC can recommend correct completion proposals even when the training model contains examples from different systems. As long as we have relevant usage contexts, CSCC can find them and can recommend completion proposals. Second, despite the considerable increase in the size of the training model, we did not notice significant improvement in performance. This seems to be consistent with Hindle et al.'s finding that the degree of regularity across project is similar to that in a single project [30].

VI. Threats to validity

There are a number of threats to the validity of this study.

First, we considered only two APIs. One can argue that the result may be different for a different framework or library. While it can be beneficial to test with additional libraries for other reasons, given that CSCC does not directly rely on these libraries, we believe that this is highly unlikely and that the results we obtain in this paper should largely carry over.

Second, we re-implement the BMN system since both the data and implementation of the technique are not available. Although we cannot guarantee that our replication of the technique does not contain any error, we've spent a considerable amount of time implementing and testing the technique to minimize the possibility of introducing error.

Third, in this study we only consider top four lines to determine the context of a method call because we assume a developer is typing code in a top-down manner, which is consistent with previous studies [12]. However, it is also possible that a developer can edit existing code, in which case we can use both top and bottom lines of a method call to create context. Supporting such alternative manners of developing code remains as future work.

VII. Conclusion

In this paper, we present a simple, efficient, scalable, context sensitive code completion technique, called CSCC. CSCC mines previous code examples to recommend completion proposals. CSCC is simple because it is based on tokenization, instead of parsing or other more advanced analysis. It is efficient and scalable due to its ways of measuring context similarity: using *simhash* first as a coarse-grained but efficient filter, and LCS/Levenshtein distance second as a refined, more accurate similarity metrics. We've compared CSCC with other state-of-the-art code completion systems using eight open source software systems that use two popular libraries. CSCC performed better than state-of-the-art static type or context-sensitive, example-based systems considered in our study. We also propose a taxonomy of method calls to identify the effect of different context elements on code completion techniques.

We envisage extending CSCC by incorporating knowledge captured by our taxonomy. Moreover, code completion operations are inherently repetitive and developers often insert the same text within a short period of time [31]. Although we did not consider method call repetitiveness in our taxonomy, we believe that this could be a potential additional information to further improve the result. Lastly, we would like to conduct a user study to evaluate the effectiveness of code completion systems.

Acknowledgements: We would like to thank Marcel Bruch and Andreas Sewe for providing useful comments and explaining APIs of Code Recommenders.

References

[1] G. C. Murphy, M. Kersten, and L. Findlater, "How are java software developers using the eclipse IDE?", IEEE Software, vol. 23, no. 4, 2006, pp. 76-83.
[2] G. Little, Greg and R. C. Miller, "Keyword Programming in Java", in Proc. ASE, 2007, pp. 84-93.
[3] D. Hou and D. M. Pletcher, "An evaluation of the strategies of sorting, filtering, and grouping API methods for Code Completion", in Proc. ICSM, 2011, pp. 233-242.
[4] R. Hill and J. Rideout, "Automatic method completion", in Proc. ASE, 2004, pp. 228-235.
[5] C. Zhang, J. Yang, Y. Zhang, J. Fan, X. Zhang, J. Zhao, and P. Ou, "Automatic parameter recommendation for practical API usage", in Proc. ICSE, 2012, pp. 826-836.
[6] D. M. Pletcher and D. Hou, "BCC: Enhancing code completion for better API usability", in Proc. ICSM, 2009, pp. 393-394.
[7] M. Mooty, A. Faulring, J. Stylos, and B. A. Myers, "Calcite: Completing Code Completion for Constructors Using Crowds", in Proc. VLHCC, 2010, pp. 15-22.
[8] S. Han, D. R. Wallace, and R. C. Miller, "Code Completion from Abbreviated Input", in Proc. ASE, 2009, pp. 332-343.
[9] A. T. Nguyen, H. A. Nguyen, T. T. Nguyen, and T. N. Nguyen, "GraPacc: A graph-based pattern-oriented, context-sensitive code completion tool", in Proc. ICSE, 2012, pp. 1407-1410.
[10] A. T. Nguyen, T. T. Nguyen, H. A. Nguyen, A. Tamrawi, H. V. Nguyen, J. Al-Kofahi, and T. N. Nguyen, "Graph-based pattern-oriented, context-sensitive source code completion", in Proc. ICSE, 2012, pp. 69-79.
[11] R. Robbes and M. Lanza, "How Program History Can Improve Code Completion", in Proc. ASE, 2008, pp. 317-326.
[12] M. Bruch, M. Monperrus, and M. Mezini, "Learning from examples to improve code completion systems", in Proc. FSE, 2009, pp. 213-222.
[13] D. Hou and D. M. Pletcher, "Towards a better code completion system by API grouping, filtering, and popularity-based ranking", in Proc. RSSE, 2010, pp. 26-30.
[14] M. Bruch, T. Schäfer, and M. Mezini, "On evaluating recommender systems for API usages", in Proc. RSSE, 2008, pp. 16-20.
[15] "The Code Recommenders" http://www.eclipse.org/recommenders/
[16] "The Vuze" hhttp://www.vuze.com/
[17] "The Subversive" https://www.eclipse.org/subversive/
[18] "The RSSOwl" http://www.rssowl.org/
[19] "The Eclipse" http://www.eclipse.org/
[20] "The NetBeans" https://netbeans.org/
[21] "Introduction to Information Retrieval", http://www-nlp.stanford.edu/IR-book/
[22] "The jEdit" http://sourceforge.net/projects/jedit/
[23] "The ArgoUML" http://argouml.tigris.org/
[24] "The JFreeChart" hhttp://sourceforge.net/projects/jfreechart/
[25] "Code Examples" http://examples.oreilly.com/jswing2/code/
[26] M. S. Uddin, C. K. Roy, K. A. Schneider, and A. Hindle, "On the Effectiveness of Simhash for Detecting Near-Miss Clones in Large Scale Software Systems", in Proc. WCRE, 2011, pp. 13-22.
[27] G. S. Manku, A. Jain and A. D. Sarma, "Detecting NearDuplicates for Web Crawling", in Proc. WWW, 2007, pp. 141-150.
[28] M. S. Charikar, "Similarity estimation techniques from rounding algorithms", in Proc. STOC, 2002, pp. 380-388.
[29] T. T. Nguyen, A. T. Nguyen , H. A. Nguyen , T. N. Nguyen, "A statistical semantic language model for source code", in Proc. FSE, pp. 532-542, 2013
[30] A. Hindle , E. T. Barr , Z. Su , M. Gabel, P. Devanbu, "On the naturalness of software", in Proc. ICSE, pp. 837-847, 2012.
[31] T. Omori, H. Kuwabara, K. Maruyama, "A study on repetitiveness of code completion operations", in Proc. ICSM, pp.584-587, 2012