# Recommending API Methods Based on Identifier Contexts

Lars Heinemann Benjamin Hummel
Technische Universität München, Garching b. München, Germany
{heineman,hummelb}@in.tum.de

## ABSTRACT

Reuse recommendation systems suggest functions or code snippets that are useful for the programming task at hand within the IDE. These systems utilize different aspects from the context of the cursor position within the source file being edited for inferring which functionality is needed next. Current approaches are based on structural information like inheritance relations or type/method usages. We propose a novel method that utilizes the knowledge embodied in the identifiers as a basis for the recommendation of API methods. This approach has the advantage that relevant recommendations can also be made in cases where no methods are called in the context or if contexts use distinct but semantically similar types or methods. First experiments show, that the correct method is recommended in about one quarter to one third of the cases.

## Categories and Subject Descriptors
D.2.13 [**Software Engineering**]: Reusable Software—*Reusable libraries*

## General Terms
Algorithms, Experimentation

## Keywords
Software Reuse, Recommendation System

## 1. INTRODUCTION

Reuse of existing code is often regarded as the best approach to achieve tremendous increases in developers' productivity, improvements of software quality, and reduced development cost [6, 7]. The goal is to integrate existing and well-tested software components, libraries, and frameworks[1] instead of reinventing the wheel by writing the code from scratch. Reaching reuse rates of 90% and more [5] can typically only be achieved by integration of multiple different libraries from different sources, accumulating to large amounts of code. The challenge for a developer is to locate a reusable piece of code for a certain task within the libraries used by the project.

As a possible solution for this task, several search engines have been proposed, such as Google Code Search[2] and Koders[3], which both work on the textual level, or Code Conjurer [4], which uses test cases for querying and retrieving code. To use a search engine, however, the developer has to actively decide to look for reuse opportunities and formulate a query based on his intent. A slightly different approach is taken by reuse recommendation systems, such as Rascal [9] or CodeBroker [12]. There, the developer is proactively and automatically informed of possible reuse options based on the current piece of code he is working on. This relieves the developer from formulating queries and can bring up possibly reusable code snippets even when the developer did not consider a search worthwhile.

**Position Statement.** Existing reuse recommendation systems work on the syntactic structure of the program; for object oriented systems this is provided by the different classes and methods and their signatures. Based on prior knowledge of which methods are often used together or in the context of certain types, the system can suggest methods based on the types and methods most recently used by the developer. This obviously works well for highly specialized data types and APIs, where only a couple of methods are applicable to a certain kind of object. However, this approach is weak when confronted with more general types that are used in a large variety of different contexts. As an example, consider the following Java code snippet from an open source system:

```
if (angle != getAngle()) {
  float angleDelta = angle - getAngle();
  super.setAngle(angle);
```

The only type used is *float* and only getter/setter methods from the project's own code are called. As the number of methods that can work on *float*s is huge, the system is likely to recommend methods that are not relevant for the developer's current task. However, looking at the identifiers it is obvious that the program is dealing with angles, so suggesting trigonometric functions, such as Math.sin(), could be helpful. Based on the observation, that about 70% of a system consist of identifiers [2] which carry crucial information about a developer's intent, we suggest to base recommendation of methods on the content of these identifiers.

[1]While all of these have a slightly different focus and intent, we will refer to all of these as *libraries* here.

[2]http://www.google.com/codesearch
[3]http://www.koders.com

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
*SUITE* '11, May 28, 2011, Waikiki, Honolulu, HI, USA
Copyright 2011 ACM 978-1-4503-0597-6/11/05 ...$10.00.

**Contribution.** We describe an approach for mining common sets of terms found in the identifiers preceding a method call. Based on this data, our system recommends methods using the identifiers from the code a developer is currently working on. We report on preliminary experiments showing that our approach is able to recommend the "right" method in about one quarter to one third of all cases.

## 2. APPROACH

The proposed approach is based on the fundamental heuristic that code fragments using similar terms within the identifiers also reuse similar methods. It consists of an upfront mining process that extracts knowledge about the association between terms and reused methods from existing software systems. The obtained data, which is stored on disk, is then used for automatically recommending methods based on context information of newly developed code.

### 2.1 Association Mining

The association mining process extracts associations between combinations of terms and methods from existing code. This is done by parsing the source code files of a software system and analyzing each method call[4]. We utilize the Eclipse Java compiler (ECJ) to obtain the abstract syntax tree (AST) for a Java class. After parsing, the AST is traversed and each method call located within a method body is analyzed. For each method call, we attempt to identify a configurable number of non-blank, non-commented lines within the same method body preceding the considered method call. This is obviously not possible for all method calls in the source code (*e. g.*, method calls in the first line of a method body). From the preceding lines and the line with the method call (up to the position of the call itself), we determine all identifiers used. The identifiers can be formed of several words (*e. g.*, sortedCustomerList). In Java there is a common convention to use the camel case notation for compound identifiers. We use this convention to split identifiers into distinct words. Identifier parts consisting of a single character are discarded. We furthermore process the split identifiers by performing word stemming, *i. e.*, reducing inflected words to their stem. This has the advantage that syntactically distinct words that refer to the same concept are treated equally[5]. The result of this procedure is a set of terms. Consider the following code snippet as an example:

```
try {
  readFile();
}
catch (IOException e) {
  String message = e.getMessage();
  JOptionPane.showMessageDialog(null,
    message);
}
```

For the method call `JOptionPane#showMessageDialog`, the set of context identifiers in the 3 preceding lines is:

`{IOException, e, String, message, getMessage}`.

After splitting and stemming the set of terms is:

`{io, except, string, messag, get}`[6].

[4]including constructor calls

[5]For instance the words "reading", "reader", "read" would all be reduced to their common stem "read"

[6]the single-character identifier e was discarded

$$\begin{array}{ccccccccl} \text{read} & \text{file} & \text{io} & \text{except} & \text{string} & \text{messag} & \text{get} & \ldots & \\ 1 & 1 & 1 & 1 & 1 & 1 & 0 & \ldots & \text{Throwable\#getMessage} \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & \ldots & \text{JOptionPane\#showMessageDialog} \\ \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \end{array}$$

**Figure 1: Association matrix**

The approach constructs a matrix from the method calls that associates a combination of terms with a method call. Each call is represented by one row within this matrix. The row consists of a binary vector denoting the combination of terms used in the identifiers in the vicinity of the method call and the name of the method that was called. The term vector has as many components as there are terms used in the identifiers in the complete source code. A component of "1" indicates that the term occurred and a "0" that it did not occur respectively. Figure 1 shows the association matrix for the two Java API calls in the example code snippet.

### 2.2 Method Recommendation

To use the association matrix for recommending methods, our implementation answers queries formed of a set of terms extracted from the context of the current cursor position by returning a set of method recommendations. This is achieved by transforming the set of terms into its corresponding binary vector representation. Then, its nearest neighbors, with a certain maximum distance in the vector space are determined. The distance of two sets of terms corresponds to the Hamming distance of their binary vectors, *i. e.*, the number of components at which the two binary vectors differ. The methods of these neighbors are then recommended in the order of their number of occurrences among these neighbors. The maximum distance is increased until the desired number of method recommendations is reached.

## 3. EXPERIMENTAL RESULTS

**Setup.** In order to evaluate our approach, we assessed how well it can be used to recommend methods from the Java API, *i. e.*, we considered only method calls to the Java API. We used 12 open source systems of various types and sizes known to us from previous studies with a total of 3 MLOC for the experiments.

We implemented the association matrix construction as well as the evaluation on top of our open source quality analysis framework ConQAT[7], which provides—among others—basic functionality for static code analysis.

To evaluate our approach, we analyzed for a given method call in a system and its context, how well our approach can predict the method call from the terms contained in the identifiers in the preceding lines of the method call. Thereby we experimented with different values for the *look back i. e.*, the number of lines from the context considered for extracting the identifiers. We queried the matrix for the 5 best recommendations[8]. The recommendation set was considered

[7]`http://www.conqat.org`

[8]In our opinion, a developer is willing to inspect 5 recommendations, even though the approach is not able to recommend useful methods in all situations

**Table 1: Recommendation Baseline**

| Project | CR |
|---|---|
| DrJava | 14% |
| Freemind | 15% |
| HSQLDB | 25% |
| Jabref | 15% |

correct, if the method actually used was among the recommended methods. More formally, we computed the fraction of methods that are correctly recommended ($CR$) as follows.

Let $MCWC$ be the set of all method calls with an appropriate context in the source code of the system, $method(c)$ the method targeted by the method call $c$, $query(T)$ the result of a query to the association matrix with the set of terms $T$ and $terms(c)$ the set of terms in the context of the method call $c$. $CR$ is then given by:

$$CR = \frac{|\{c \in MCWC | method(c) \in query(terms(c))\}|}{|MCWC|}$$

**Recommendation Baseline.** As a comparison, we computed a baseline for the ratio of correct recommendations for a trivial approach that for a given project always recommends the 5 methods used most frequently within that project. We considered all method calls with a context of at least one line. The results are shown in Table 1.

**Cross Project Recommendation.** We evaluated how well the association matrix constructed from a set of projects can be used to recommend methods for a different project. From the 12 study objects, we chose 4 projects for evaluating cross project recommendation. For each of the 4 projects, we constructed the association matrix from the other 11 projects. Table 2 shows the results of the evaluation. The column *LB* denotes the look back in lines. *OMC* represents the overall number of method calls to the Java API in the project. *MCWC* contains the subset of those method calls with an appropriate context where our approach could make recommendations. Column *DM* shows the number of distinct methods targeted by these method calls, which gives an intuition about the difficulty of the recommendation problem. The fraction of the method calls with an appropriate context that were correctly recommended is given in column *CR*.

**Table 2: Cross Project Recommendation**

| Project | LB | OMC | MCWC | DM | CR |
|---|---|---|---|---|---|
| DrJava | 1 | 21,090 | 16,167 (77%) | 1,747 | 31% |
| | 2 | | 15,244 (72%) | 1,656 | 29% |
| | 3 | | 14,347 (68%) | 1,573 | 26% |
| Freemind | 1 | 8,725 | 6,726 (77%) | 1,255 | 32% |
| | 2 | | 6,090 (70%) | 1,122 | 29% |
| | 3 | | 5,588 (64%) | 1,061 | 27% |
| HSQLDB | 1 | 9,735 | 8,404 (86%) | 1,006 | 33% |
| | 2 | | 8,113 (83%) | 964 | 32% |
| | 3 | | 7,833 (80%) | 931 | 30% |
| Jabref | 1 | 21,350 | 17,020 (80%) | 1,471 | 39% |
| | 2 | | 16,167 (76%) | 1,387 | 37% |
| | 3 | | 15,375 (72%) | 1,330 | 34% |

**Table 3: Intra Project Recommendation**

| Project | LB | OMC | MCWC | DM | CR |
|---|---|---|---|---|---|
| DrJava | 1 | 9,993 | 7,880 (79%) | 1,115 | 39% |
| | 2 | | 7,462 (75%) | 1,049 | 38% |
| | 3 | | 7,052 (71%) | 1,002 | 36% |
| Freemind | 1 | 5,427 | 4,239 (78%) | 929 | 26% |
| | 2 | | 3,805 (70%) | 814 | 23% |
| | 3 | | 3,488 (64%) | 771 | 19% |
| HSQLDB | 1 | 6,031 | 5,331 (88%) | 755 | 38% |
| | 2 | | 5,158 (86%) | 722 | 36% |
| | 3 | | 5,004 (83%) | 702 | 35% |
| Jabref | 1 | 9,654 | 8,006 (83%) | 1,076 | 43% |
| | 2 | | 7,592 (79%) | 1,022 | 43% |
| | 3 | | 7,226 (75%) | 995 | 40% |

**Intra Project Recommendation.** We additionally evaluated the quality of the recommendations within a single project by building the association matrix from a subset of the project and predicting the method calls for the remaining part of the project. This is a relevant scenario since in many projects an existing code base is extended and maintained. Intuitively it can be expected that there is more homogeneity in the term-method associations within a single project. We therefore expect better recommendation results. We randomly selected half of the files of the project and computed recommendations for all method calls in the other half of the files. Table 3 shows the results of the evaluation for intra project recommendation.

## 4. DISCUSSION

The results show, that the correct method is in the 5 methods recommended by our approach in one fourth to one third of the cases. Furthermore, the results are significantly improved compared to the baseline heuristic of always recommending the 5 most frequently used methods. The exception is HSQLDB, where the gain is less pronounced. This can be explained by the distribution of method calls in HSQLDB that already leads to a high baseline value.

When learning the association matrix from a part of the project itself, the relative amount of correct recommendations increases in most cases. This is expected, as the vocabulary and naming conventions are assumed to be more consistent within a single project than between projects. Interestingly, for the *Freemind* project, the opposite is the case. One explanation is the smaller size of the project that leaves less examples to extract the associations from.

One central parameter is the number of previous lines we use as context. Interestingly, using 2 or even 3 lines of context leads to lower numbers of correctly recommended methods. This seems counterintuitive at first, as using more information is expected to improve the results. Currently, our best guess is that the additional lines of context disturb the prediction as often as they aid it. Information from the lines before might be often unrelated to the current method call. Overall, between 64% and 88% of the method calls provide sufficient context to allow our approach to be applied. This indicates that even for a required context of 3 lines, there are sufficiently many locations where our approach is applicable.

## 5. RELATED WORK

Mapo [13] mines API usage patterns from open source repositories and recommends code snippets that illustrate usage scenarios of a queried API method. The mined patterns describe API methods that are frequently called together in a sequence. While Mapo recommends API usage examples based on an API method that is already known to the developer, our method can suggest methods from APIs yet unknown to the developer.

Rascal [9] stores the method usages of classes in a database and uses this information to recommend methods for a currently developed class that is similar to classes in the database. It employs different collaborative filtering approaches to compute the similarity between classes. While this approach is based on similarity between the method calls of whole classes, our method uses the identifiers in the context of a few source lines preceding a method call.

Strathcona [3] suggests relevant code in a repository of code examples. This tool automatically extracts structural properties like inheritance relations or method calls of the code currently developed and matches them with those of the code in the repository. While Strathcona recommends code examples, our method suggests API methods.

CodeWeb [10] mines patterns of library reuse based on association rules known from shopping basket analysis. An example of a mined rule would be that classes deriving from a certain class usually override a certain method of the derived class. While CodeWeb is a browser for reuse patterns, our method aims at recommending API methods.

CodeBroker [12] proactively suggests useful yet unknown methods within the Emacs editor by analyzing the program under development and performing a similarity analysis with methods in a repository. The similarity is defined in terms of programmer documentation and method signatures. When the developer enters a JavaDoc comment and (optionally) declares the signature of the desired method, the tool automatically creates a query from this information that is used to locate methods that provide the required functionality. While CodeBroker requires the desired functionality to be described (comment, signature) for making recommendations, our approach uses the context to suggest methods that are likely to be needed next.

ParseWeb [11] and Prospector [8] can answer queries of the form *Source object type* $\rightarrow$ *Destination object type* by suggesting method sequences that take the source object as input and yield the destination object as a result. While these tools require the source and destination type of the desired method to be known, our method can suggest methods with yet unknown or unused types.

Code Conjurer [4] automatically infers queries for reusable classes from the tests written in advance to the implementation according to the test-driven development technique. In contrast to our approach, this method requires test cases to be written for the retrieval of reusable functionality.

Bruch et al. [1] propose an approach for improving the relevance of suggestions made by code completion systems. While they use an approach very similar to ours based on the k-nearest-neighbor algorithm to determine the similarity of code contexts, they consider method calls as the context. Moreover, they focus on recommending relevant methods from a particular class requiring the object on which the method is called to be known.

## 6. CONCLUSION AND FUTURE WORK

We described a reuse recommendation system based on the content of identifiers. From patterns mined from other systems or existing parts of the system, our tool infers methods to be used given only the identifiers from the code the programmer is currently working on. Our experiments indicate, that our approach is able to suggest the correct method in about every fourth to third case (when an appropriate code context is available).

More research is required to understand for which methods our approach works well or not. It is also open, whether our approach is correct for the same cases as those based on program structure, or if a combined approach could lead to better results. Other ideas for improving the results are to respect synonyms when searching methods, for example by integrating the WordNet[9] ontology, or applying different comparison strategies for the context, such as weighting identifiers by distance from the method call. Finally, the recommendations should be evaluated in experiments with developers. Therefore, it might be relevant to exclude methods well-known to a developer from the recommendations.

## 7. REFERENCES

[1] M. Bruch, M. Monperrus, and M. Mezini. Learning from examples to improve code completion systems. In *ESEC-FSE'09*, 2009.

[2] F. Deissenboeck and M. Pizka. Concise and consistent naming. *Software Quality Journal*, 14(3):261–282, 2006.

[3] R. Holmes and G. C. Murphy. Using structural context to recommend source code examples. In *ICSE'05*, 2005.

[4] O. Hummel, W. Janjic, and C. Atkinson. Code conjurer: Pulling reusable software out of thin air. *IEEE Software*, 25(5):45–52, 2008.

[5] I. Jacobson, M. L. Griss, and P. Jonsson. Making the reuse business work. *IEEE Computer*, 30(10):36–42, 1997.

[6] C. Krueger. Software reuse. *ACM Comput. Surv.*, 24(2):131–183, 1992.

[7] W. Lim. Effects of reuse on quality, productivity, and economics. *IEEE Software*, 11(5):23–30, 2002.

[8] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman. Jungloid mining: helping to navigate the API jungle. *ACM SIGPLAN Notices*, 40(6):48–61, 2005.

[9] F. Mccarey, M. Cinnéide, and N. Kushmerick. Rascal: A recommender agent for agile reuse. *Artificial Intelligence Review*, 24:253–276, 2005.

[10] A. Michail. Data mining library reuse patterns using generalized association rules. In *ICSE'00*, 2000.

[11] S. Thummalapenta and T. Xie. Parseweb: a programmer assistant for reusing open source code on the web. In *ASE'07*, 2007.

[12] Y. Ye and G. Fischer. Information delivery in support of learning reusable software components on demand. In *IUI'02*, 2002.

[13] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei. MAPO: Mining and recommending API usage patterns. In *ECOOP'09*, 2009.

[9]`http://wordnet.princeton.edu`