Identifier-Based Context-Dependent API Method Recommendation

Lars Heinemann Veronika Bauer Markus Herrmannsdoerfer Benjamin Hummel
Technische Universität München, Germany
{*heineman, bauerv, herrmama, hummelb*}*@in.tum.de*

***Abstract*—Reuse recommendation systems support the developer by suggesting useful API methods, classes or code snippets based on code edited in the IDE. Existing systems based on structural information, such as type and method usage, are not effective in case of general purpose types such as String. To alleviate this, we propose a recommendation system based on identifiers that utilizes the developer's intention embodied in names of variables, types and methods. We investigate the impact of several variation points of our recommendation algorithm and evaluate the approach for recommending methods from the Java and Eclipse APIs in 9 open source systems. Furthermore, we compare our recommendations to those of a structure-based recommendation system and describe a metric for predicting the expected precision of a recommendation. Our findings indicate that our approach performs significantly better than the structure-based approach.**

***Keywords*-software reuse; recommendation system; identifier; data mining**

I. Introduction

Code reuse is known to have positive effects on productivity, overall product quality, and time to market [1], [2]. A main challenge for effective reuse is the retrieval of reusable entities, which has been an active area of research since the late 1990s [3]. Today, the Internet provides a tremendous amount of freely reusable code and thereby serves as a rich reuse repository [4]. It is even said to be emerging as "a de facto standard library for reusable assets" [5]. With this ubiquitous code repository, reuse rates[1] of 40% and more are achieved in many open source Java projects [6]. While this represents a big opportunity for productivity improvement, lack of accessibility is preventing to unlock the full potential promised by reuse. Often, libraries and frameworks are large and complex. For instance, version 1.7 of the Java API lists more than 4,000 API types. Within these large sets of reusable entities, identifying those relevant for the task at hand is a real challenge even for experienced developers. To ease this difficulty, numerous code retrieval approaches have been proposed [4], [7]–[9]. Even though they report improvements in API accessibility and thus encourage reuse, they still require active user interaction in terms of formulating a query. Two problems arise as a consequence: Firstly, no matter how advanced the tool, the developer must already have an idea of what to look for. Secondly, developers are forced to interrupt their workflow, which is likely to disturb their concentration and focus on the current problem, consequently decreasing their efficiency [10].

[1]ratio between reused code and total amount of code of a software system

To alleviate these problems, reuse recommendation systems have been proposed. These systems automatically derive queries from the current development context and propose reusable code useful for the task at hand. Consider the following code snippet from an open source Java system:

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

As the code deals with angles, trigonometric functions, such as Math.sin(), could be recommended to the user. Existing reuse recommendation systems utilize structural program information, *e. g.*, the types and methods used, to infer what functionality might be needed [11]–[13]. However, cases where no methods or types are used in the context or only general purpose types are used, are not supported well by these approaches. In our example snippet, the only type involved is float and the only methods are simple getters and setters. Thus, when relying only on structural information, a reliable recommendation is hard to determine.

In preliminary experiments, we showed that identifiers, *i. e.*, the names of the variables, types and methods in a program chosen by a developer, are a promising source for mining term-method associations that can be used to build a method recommendation system [14]. In this paper, we evaluate the impact of several modifications to the basic algorithm and perform a quantitative comparison to an existing structure-based approach. We evaluate our approach for recommending methods from the Java and Eclipse APIs and present initial results on the potential of a hybrid approach using both structural and identifier information.

Research problem. Current reuse recommendation systems use structural information such as method or type usage in the code to derive recommendations. However, if no method calls exist in the code of the current context or only general purpose types are used, these approaches are not effective. Identifiers, on the other hand, are typically available in any context, and as identifiers embody valuable knowledge about the intent of a programmer [15], they represent a promising basis for recommendation systems. It needs to be determined how well information from identifiers can be used for building a recommendation system and how the recommendation quality compares to that of existing structure-based approaches. Furthermore, it is an open question whether a hybrid approach has the potential of outperforming stand-alone structure-based or identifier-based approaches.

Contribution. We describe an approach that mines the intentional knowledge embodied in the identifiers of existing source code and uses an index of this information to recommend API methods based on the code being developed. We evaluate the precision of the approach by recommending Java API methods for 6 open source Java systems and Eclipse API methods for 3 open source Eclipse-based projects. Furthermore, we analyze the impact of several algorithm parameters on the recommendation results. Finally, we quantitatively compare our approach to an existing structure-based recommendation system and investigate to which degree the expected precision of both approaches can be estimated for a given context.

II. Related Work

Two major directions have been proposed to increase API accessibility, thus fostering reuse: classic code retrieval and code recommendation systems. Code retrieval allows to query code from repositories. Prominent examples are keyword-based code search engines like Google Code Search[2] or Koders[3]. Advanced approaches use signatures [16], specifications [17], test cases [4], or combinations thereof [18] to search for reusable entities. These methods, however, require users "to derive abstractions of what they actually want in order to find the artifacts that are potentially useful" [3]. On the language level, the "vocabulary problem" [19] often hampers the effectiveness of the listed tools as developers unfamiliar with an API may query in a terminology differing to the one employed by the API. Recent approaches address this issue, enabling free-form natural language queries: SNIFF [7], a Java search engine, enriches the API with its documentation. Apatite [8], provides an interface for associative browsing of the Java and Eclipse APIs and aids the developer by providing also API items frequently related to their query. Hill et al. [9] propose an approach for contextual code search, which enriches query results with natural language phrases drawn from the context of the proposed method.

Despite their increase in effectiveness, code retrieval approaches fail to overcome an important obstacle: the user has to abandon the current task to actively formulate a query, which disrupts his workflow. Code recommendation systems address this issue by using the current development context to automatically extract queries and recommend code entities that may be useful for the task at hand. This can even happen proactively, *i. e.*, without the user anticipating the existence of a relevant code entity.

In the remainder of this section, we focus on code recommendation systems which can be categorized as follows:

"Classic" IDE code completion offers a list of available variables, types and methods. Partially entered names are used to narrow the proposed completions. The focus is on syntactical applicability and visibility of the suggestions. The order of the proposals is usually regarding the type of item (*e. g.*, method, variable, type) and furthermore alphabetical. The quality of the suggestions heavily depends on the programming language. Strongly typed languages typically allow for better suggestions, since the type system allows to infer what suggestions are applicable.

Enhanced code completion aims at improving code completion systems by producing more relevant completion proposals. In [20], Bruch et al. introduce an example based code completion system. They enhance current code completion systems by making context-sensitive method recommendations. Their approach is based on mining knowledge from an example code base. They present three different methods for using the information in the code repository: frequency of method calls, association rule mining and a modification of the k-nearest-neighbor algorithm. While they employ a method similar to ours, they use method calls as the context. Moreover, they recommend only methods applicable to a variable of a class type. In contrast, since we use the identifiers as the context, we can recommend methods of classes yet unused by the developer.

Code example recommendation systems suggest complete code examples. Bajracharya et al. [21] propose *structural semantic indexing* which utilizes API usage similarities extracted from code repositories to associate words with source code entities. Their goal is to improve the retrieval of API usage examples from code repositories by addressing the vocabulary mismatch problem. This is done by sharing terms among code entities that have similar usage of APIs and are thus functionally similar. Mapo [22] mines API usage patterns from existing source code. The patterns are given by methods that are frequently called together and that follow certain sequential rules. The patterns are used for answering queries for methods by returning example snippets illustrating the usage of that method. Strathcona [23] is a tool for retrieving useful code snippets from a repository of code examples. It extracts the structure of the code under development and matches it to the code in the repository. The repository can be created automatically from existing software systems. The authors developed several heuristics for structure matching of code, including inheritance and call/use relations. While these tools suggest code examples, our approach recommends API methods.

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

API method recommendation systems propose methods or method sequences from APIs based on different aspects of the development context. CodeBroker [24] infers queries for reusable components from Javadoc comments and signatures of the code currently being developed. It uses *latent semantic indexing* for matching the comments in the code against those of repository code. Signature matching is used to ensure the syntactic compatibility with the code context. While our approach uses the code context to make recommendations, CodeBroker requires the desired functionality to be described in terms of comments and/or signatures. Parseweb [12] and Prospector [13] can answer queries of the form *source class type* $\rightarrow$ *destination class type*. The result is a method sequence that transforms an object of the source type to an object of the destination type. As opposed to our approach, these tools require the source and destination type of a required method sequence to be known. Rascal [11] is the approach most similar to ours. It suggests API methods based on a set of already employed methods within a class. It uses *collaborative filtering* (CF), whereby classes are interpreted as users and the called methods are treated as items. While we use the identifiers in the vicinity of a method call to associate terms with methods, Rascal utilizes the similarity of method usage of whole classes.

Class recommendation systems suggest useful classes from APIs or code repositories. Javawock [25] employs an approach similar to Rascal. The main difference is that the items for the CF are API classes instead of methods. The authors also investigate *item-based* CF in addition to *user-based* CF done by Rascal. Code Conjurer [26] automatically retrieves reusable components based on JUnit test cases written in advance according to the test-driven development methodology. The test cases are executed on candidate components that match the required signature derived from the usage in the test code. Components passing the tests are then returned as recommendations. While these tools recommend classes, our approach suggests API methods.

III. Terms & Definitions

Development context. The development context refers to the code edited within an IDE. In the scope of this paper, this context is given by the source code preceding a cursor position within a source code file.

API method. An API method is a method (static or non-static) that is declared in a type that belongs to an API, such as the Java API, and has protected or public visibility. An API method is uniquely identified by its signature, *i. e.*, methods with the same name but different parameters are considered as distinct methods.

Recommendation set. A recommendation set is a set of API methods recommended in a given development context.

Recommendation rate. The recommendation rate denotes the fraction of correct recommendation sets produced by a method recommendation system when trying to predict method calls in existing code from the context of the method call. A recommendation set is considered *correct*, if the method actually employed is in the recommendation set.

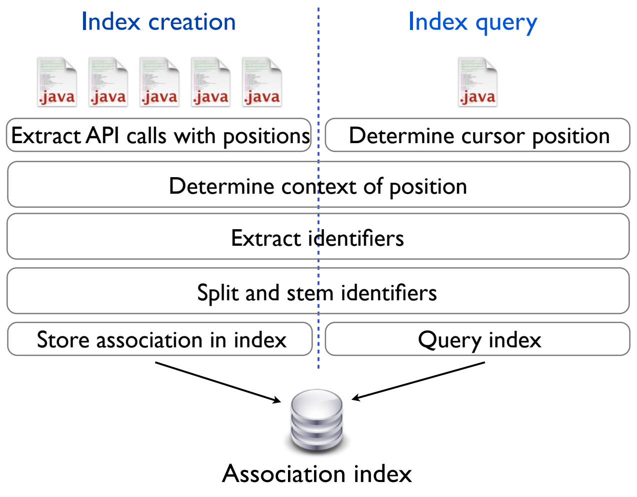
Index creation
Index query
.java
.java
.java
.java
.java
.java
Extract API calls with positions
Determine cursor position
Determine context of position
Extract identifiers
Split and stem identifiers
Store association in index
Query index
Association index

Figure 1. Overview of our approach

IV. Approach

This section describes our API method recommendation approach, which uses data mining for learning term-method associations that are used for recommending API methods during the development of new code.

The approach has a training phase that builds an association index by analyzing existing software systems. This index is then used to answer queries that are formed of a set of terms extracted from the development context. Both, the up-front index creation and the index query, share a number of common processing steps as illustrated in Figure 1.

We implemented the approach in Java on top of the open source quality analysis framework ConQAT[4], which provides basic functionality for static code analysis. Our current implementation is targeted at Java programs, but could be adapted to other programming languages.

A. Index Creation

The index creation analyzes API method calls in Java classes. We build and traverse the abstract syntax tree (AST) of a class and process all API method calls. For each method call, we extract a configurable number of identifiers preceding the method call (called *lookback*). To avoid unrelated identifiers, we consider only identifiers within the same method body. This also means that for method calls close to the beginning of a method body, it is possible that not enough identifiers can be extracted and thus no context can be determined. The extracted identifiers are split into words according to the camel case notation, which is

[4]http://www.conqat.org/

recommended by the official Java Code Conventions [27]. Identifier parts that only consist of a single character are discarded. The split words are reduced to their stem by using an English stemmer[5]. Multiple occurrences of the same term are treated as one by the term extraction process. We store an entry in the index that associates the set of terms with the API method that was called. The method is represented by its signature consisting of the fully qualified name of its declaring type[6], the method name, and the list of parameter types. The index consists of a list of these association entries.

Consider the following code snippet as an example:

```
1 try {
2   readFile();
3 }
4 catch (IOException e) {
5   String errorMessage = e.getMessage();
6   JOptionPane.showMessageDialog(null,
        errorMessage);
7 }
```

From this snippet, using a lookback of 5 identifiers, the following association entry would be extracted for the API method call in line 6:

$$\{io, except, string, error, messag, get\} \rightarrow$$
$$javax.swing.JOptionPane\#showMessageDialog($$
$$Component, Object)$$

The index creation step has several parameters, which are discussed in the following.

Lookback (1,2,3,...). The number of distinct identifiers preceding a method call that are considered for the extraction of terms can be configured with the parameter *lookback*.

Stopwords (preserved, removed). In information retrieval, stopwords are words that occur very frequently and are not useful to find documents depending on a search query. These words are typically not indexed by search engines. Examples include articles and prepositions [28]. The parameter *stopwords* of our approach denotes whether these words shall be removed from the set of terms associated with an API method. We used the list of 119 stopwords from [29].

Keywords (included, excluded). Since keywords carry information about the structure of the code, they can serve as a simple extension to our approach for considering structural information in addition to the identifiers. For instance, the keyword `catch` indicates an exception handler and can therefore add important information to a method association in our index. The parameter *keywords* specifies whether keywords that occur in the context of the method call should be considered in addition to identifiers. Technically, we wrap the keywords with brackets (which are non-identifier characters) and treat them just as normal terms during subsequent processing. For instance, the keyword `catch` is transformed to `<catch>`. Due to the brackets, we obtain a *unique term* and avoid a meaningless relation to an equal term embodied in an identifier.

B. Index Query

A query to the index consists of a set of terms extracted from the code preceding the current cursor position within a source code editor. The extraction of the terms is done as in the index creation. From the cursor position, the context is determined by analyzing the preceding identifiers according to the configured lookback. The identifiers are split and stemmed which results in the set of terms for the query.

For a query to the index, given by a set of terms, we determine those association entries in the index whose term set is most similar to the query term set. For this, a notion of *similarity* between sets of terms is required. Our approach uses the *Jaccard* similarity known from data mining [30].

The Jaccard similarity (js) is defined for arbitrary sets and is given by the ratio between the size of the intersection and the size of the union of the sets:

$$js(T_1, T_2) = \frac{|(T_1 \cap T_2)|}{|(T_1 \cup T_2)|}$$

The following example illustrates the computation of the similarity measure for two exemplary sets of terms. Given $T_1 = \{file, input, read\}$ and $T_2 = \{file, write\}$, the value of the Jaccard similarity is computed as follows:

$$js(T_1, T_2) = \frac{|\{file\}|}{|\{file, input, read, write\}|} = 0.25$$

Based on this similarity measure, a set of method recommendations is built by successively considering the most similar index entries with decreasing similarity. For each similar entry, the associated method is added to the recommendation set until it contains a configurable yet fixed number of recommendations. Since different term sets can be associated with the same method, we may have to consider more entries than the number of required recommendations. Moreover, since multiple entries can have the same similarity to the query, it can occur that we cannot add all methods associated with these entries to the recommendation set because we already have reached the desired amount of recommendations. In that case we have to arbitrarily choose from the entries with equal distance until we have the required number of recommendations.

V. Reimplementation of Rascal

For a quantitative comparison to our approach, we chose Rascal [11] as a representative for structure-based approaches. It is most similar to our approach, since it also

[5]The English language represents parts of speech, tense, and number by inflected (i.e. morphologically varied) words. Stemming is used in information retrieval to match queries also to documents containing derived words [28]. For instance, the word *read* shall be also matched with the word *reading*.

[6]in case of inheritance, the most "specific" type in the inheritance hierarchy declaring the method is considered

recommends single API methods based on the development context. Instead of identifiers in the context, it uses the methods already employed in a class to derive what method might be needed next. Since Rascal is not publicly available and we could not obtain the implementation from the authors, we carefully reimplemented the approach as described in their paper. In this section, we briefly describe their approach and present the results of an evaluation of our reimplementation.

A. Approach

Rascal uses collaborative filtering which is used to predict the preferences of a user regarding how they “like” a particular item based on what other items the user likes and what items similar users like. Adopted for API method recommendation, Rascal interprets Java classes as users and API methods as items. A user-item preference database is created from existing software that stores for each class the API method calls it contains. The method usage of a class is modeled as a vector in the space $\mathbb{N}^n$. Each component of a vector represents as a natural number how often a particular method was called by the class and thus n is the overall number of distinct methods used collectively by all classes. For computing a recommendation set of methods, a set of already employed methods in a class under development is matched against the database. Depending on a query vector, they compute the nearest neighbors with different approaches, of which *vector similarity* performs best. The vector similarity of two users is computed as the cosine of the angle formed by their vectors. For the recommendation, they use content-based filtering to predict how a query user likes a particular item. They determine how a query user likes the items that are collectively used by its nearest neighbors. The recommendation set then consists of the 5 items liked most.

B. Reevaluation

To assess the validity of our reimplementation, we first attempted to reproduce the results reported by McCarey et al. [11] by reenacting their evaluation setup as closely as possible. They tried to predict method calls to the Swing API in classes taken from 30 GUI applications. Since they did not report what exact applications were used in their case study, we used 5 open source Java applications from our study objects (see Table I) that use the Swing API. We created the user-item preference database from them. As in the paper from McCarey et al., we consecutively removed Swing methods from the end of the class and queried the index with the sequence of the remaining Swing methods in the class. The recommendation set was considered correct, if the removed method was among the recommendation set. McCarey et al. report a recommendation rate of 43% achieved in their experiment. The user-item preference database created in their experiment contained 228 classes with calls to 761 distinct Swing methods.

The user-item preference database created for the 5 Java applications in our reevaluation contained 1,014 classes with a total of 22,963 calls to 1,942 distinct Swing methods. The average fraction of correct recommendations for the systems ranged from 40% to 44%. This indicates that our reimplementation of Rascal achieves recommendation rates close to the number reported by McCarey et al. for the vector similarity based implementation (43%). Thus, we are confident that we correctly reimplemented their approach.

VI. Case Study Design

A. Research Questions

RQ1: How do the parameters impact the recommendation rate? We investigate how the ratio of correct recommendations depends on the parameters of our approach. The best parameter settings are selected to optimize the system's performance and used as parameter configuration for the following research questions.

RQ2: How does ignoring well-known methods affect the recommendation rate? For employing a recommendation system in practice, recommending API methods that are already well-known to a developer is not desirable. We analyze how the rate of correct recommendations is affected when ignoring well-known methods in index creation and index query.

RQ3: How does the approach perform for APIs other than the Java API? To assess the transferability of our approach, we determine the quality of the recommendations for the Eclipse API.

RQ4: How does our approach compare to a method usage-based approach? To further evaluate our approach, we analyze how the recommendation results compare to those from the method usage-based approach Rascal. Furthermore, we investigate the potential of a hybrid approach.

RQ5: Can we use the similarity measure for deriving a confidence level for recommendations? We investigate whether the similarity measures of the approaches can be used for deriving a meaningful confidence level for a recommendation set.

B. Study Objects

Table I lists the study objects that we used for our case study together with a set of metric values. The column *MCAPI* denotes the total number of method calls to the studied API. The column *DM* shows the overall number of distinct API methods called within the source code. To put the measurements of the recommendation rates into perspective as well as to be able to assess the difficulty of the recommendation problem, we also computed a baseline recommendation rate for a trivial approach that always recommends the 5 API methods used most frequently within that project. The values of the baseline rate are presented in column *BLRR*.

Table I
THE STUDIED JAVA AND ECLIPSE RCP APPLICATIONS

Studied API	System	Version	Description	LOC	MCAPI	DM	BLRR
Java	DrJava	stable-20100913-r5387	Java Programming Environment	160,256	21,090	2,026	11.8%
	FreeMind	0.9.0 RC 9	Mind Mapper	71,133	8,725	1,439	12.9%
	HSQLDB	1.8.1.3	Relational Database Engine	144,394	9,735	1,100	24.0%
	JabRef	2.6	BibTeX Reference Manager	109,373	21,350	1,691	18.1%
	JEdit	4.3.2	Text Editor	176,672	17,341	1,934	10.5%
	SoapUI	3.6	Web Service Testing Tool	238,375	24,659	2,500	11.3%
Eclipse	MyTourbook	11.3	Bike Tour Visualization and Analysis Tool	238,963	18,865	1,160	12.3%
	Rodin	1.4.0	Event-B Modeling / Verification Environment	273,080	5,924	1,122	7.5%
	RssOwl	2.0.6	RSS / RDF / Atom News Feed Reader	174,643	12,774	1,199	12.6%

Java API. For RQ1, RQ2, RQ4 and RQ5, we studied the Java API. We used 6 popular Java projects of different application types from the open source project repository SourceForge[7]. All projects were among the 100 most downloaded Java applications with the development status Production/Stable[8].

Eclipse API. For RQ3, we studied the API provided by Eclipse for building Rich Client Platform (RCP) applications. We used 3 open source Eclipse RCP applications. To be representative, we chose applications from completely different domains—geovisualization, software modeling, and news reading. Since we also needed the binaries of the applications for our analyses, we were restricted to applications where we could easily download or compile the binaries. We considered as API all callable methods that are defined by classes whose full qualified name starts with `org.eclipse`.

C. Design and Procedure

To evaluate the recommendations, we "remove" method calls from existing software systems and use the recommendation system to "guess" the removed method call from its context—in case of our approach given by the identifiers preceding the method call.

We measured the suitability of the recommendations with the recommendation rate, which is computed as follows. Let $MCWC$ be the set of method calls with a non-empty context, *i. e.*, where the recommendation system is applicable and can make a recommendation. Let $m(c)$ be the method targeted by the method call c, ctx the function that yields the context for a given method call and $query$ the function that returns a set of recommended methods for a given context. The recommendation rate is then given by:

$$RR = \frac{|\{c \in MCWC \mid m(c) \in query(ctx(c))\}|}{|MCWC|}$$

Intuitively, the recommendation rate is the fraction of method calls that can be "predicted" by the recommendation system based on the context of each method call.

We mined the association index and the user-item preference database respectively from half of the project files and used it to predict the method calls in the other half of the files. The sets of files were determined randomly with a fixed random seed ensuring equal results in consecutive runs. The recommendation sets contained 5 methods each[9].

RQ1: Influence of parameters. To determine the impact of the algorithm parameters, we ran our analysis with different configurations. First, we determined the influence of different lookback values. We then used the best setting for this parameter to evaluate the influence of the parameters stopwords and keywords.

RQ2: Ignoring well-known methods. We analyzed how the recommendation rate is affected when a set of well-known methods is ignored completely during index creation and query. We approximated this set of well-known methods by the set of all methods of the 20 types used most frequently in 76 open source applications as reported in a study about the usage of the Java API by Ma et al. [31].

RQ3: Other APIs. To assess the transferability of our approach to other APIs, we evaluated the recommendation rate for the 3 Eclipse RCP applications in Table I.

RQ4: Comparison to Rascal. For the quantitative comparison of our approach with the reimplementation of Rascal, we evaluated the recommendation rate for the 6 Java applications and both approaches. Both approaches have different sets of cases where they are applicable. This has to be taken into account for a quantitative comparison. Figure 2 illustrates the applicability for method predictions of both our approach and Rascal. The outer rectangle *MC* denotes the set of all method calls in the classes used for evaluation. The set *MCAPI* contains all method calls to the API under consideration, which is the Java API in the comparison. As previously described, our approach needs a context to derive identifiers from, and Rascal needs at least one method call to be able to make recommendations. Therefore, there are cases where Rascal can make recommendations and our approach is unable to do so, and vice versa. This is illustrated by the two rectangles *Our approach applicable* and

[7]http://sourceforge.net/
[8]as of April 27th, 2011

[9]In our opinion, a developer is willing to inspect 5 potentially useful method recommendations. This is also the value used by McCarey et al.

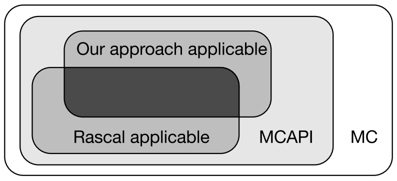

Figure 2. Applicability of approaches

Rascal applicable. We perform two comparisons. First, we evaluate the recommendation rates for both approaches for all cases where they are applicable. Second, we determine the recommendation rate for cases where both approaches are applicable, *i. e.*, we consider the method calls in the dark gray intersection in Figure 2.

We also evaluated in how many cases one of the approaches can predict a method and the other cannot. From these numbers, we computed the recommendation rate of a hypothetical ideal hybrid approach that assumes a perfect oracle which can decide which recommendation set should be returned as an answer to a query. The oracle always decides for the approach that has the correct recommendation. In case both recommendation sets are correct, an arbitrary approach is chosen. To build a real hybrid approach, we would need to implement such an oracle.

RQ5: Confidence level. To implement an oracle, we attempted to compute a confidence level for both approaches that allows us to decide for an approach depending on the query and the recommendation set produced by both tools.

To compute a confidence level for a recommendation set, we used the similarity of the nearest neighbor and most similar association entry respectively for a query. For Rascal the similarity measure is given by the vector (cosine) similarity and for our approach it is the Jaccard similarity. Formally, this can be expressed as follows. Let $nn(q)$ be the nearest neighbor or most similar index entry for a query q. Then the confidence of the recommendation for given a query q is given by:

$$c(q) = similarity(q, nn(q))$$

We related the confidence level to the recommendation rate. For both approaches, we computed the average recommendation rate over all 6 Java applications within 10 confidence intervals from 0 to 1 in steps of 0.1. The average was computed over all method calls in all applications.

VII. Results

RQ1: Influence of parameters. Table II presents the recommendation rates for lookback values ranging from 1 to 6 identifiers. The rate of correct recommendations varies only moderately for different lookback values. On average, the best recommendation rate is achieved with a lookback of 4. Consequently, for the following experiments, we use a lookback of 4 identifiers.

Table II
Recommendation rate regarding lookback

Lookback	1	2	3	4	5	6
DrJava	33.6%	38.9%	39.4%	40.5%	38.7%	39.4%
Freemind	33.1%	38.3%	39.4%	39.0%	38.8%	38.5%
HSQLDB	24.8%	34.5%	33.9%	35.9%	36.8%	38.1%
Jabref	37.0%	45.3%	45.8%	47.0%	46.1%	44.9%
JEdit	34.7%	38.8%	39.6%	39.6%	38.6%	38.2%
SoapUI	37.0%	46.8%	46.5%	45.6%	44.9%	43.9%
Average	33.4%	40.5%	40.8%	41.3%	40.7%	40.5%

Table III
Recommendation rates with stopwords preserved/removed

Project	**Stopwords**	
	preserved	removed
DrJava	40.5%	39.7%
Freemind	39.0%	39.7%
HSQLDB	35.9%	36.0%
Jabref	47.0%	46.7%
JEdit	39.6%	39.0%
SoapUI	45.6%	45.2%

Table III shows the results regarding the impact of stopwords. Removing stopwords has a very small effect on the recommendation rate. In all cases, the difference is below 0.6%. For 2 of the study objects, the rate slightly increases while for 4 the rate slightly decreases. Consequently, the stopwords parameter is set to preserved during the analysis.

Table IV presents the recommendation rates when keywords are excluded vs. included. The results show that taking keywords into account has a notable positive effect on the recommendation rates for all study objects. The analysis including keywords is performing 3.8% to 5.6% better compared to the analysis with excluded keywords.

Based on the outcomes of RQ1, the parameter configuration for the remaining analyses is as follows: a lookback of 4 identifiers, stopwords preserved, and keywords included.

RQ2: Ignoring well-known methods. Table V displays the results of the analysis when ignoring vs. including well-known methods. Ignoring well-known methods decreases the recommendation rate between 0.7% and 11.4%. We provide an interpretation of these numbers in the discussion.

RQ3: Other APIs. Table VI shows the recommendation rates of our approach applied to the Eclipse RCP applications. The resulting recommendation rate ranges from 49.4% to 67.8%, thus indicating that our approach can also be

Table IV
Recommendation rates with keywords excluded/included

Project	**Keywords**	
	excluded	included
DrJava	40.5%	44.3%
Freemind	39.0%	43.1%
HSQLDB	35.9%	41.6%
Jabref	47.0%	52.1%
JEdit	39.6%	45.1%
SoapUI	45.6%	51.2%

Table V
RECOMMENDATION RATES WITH WELL-KNOWN METHODS INCLUDED/IGNORED

Project	**Well-known methods**	
	included	ignored
DrJava	44.3%	42.8%
Freemind	43.1%	32.2%
HSQLDB	41.6%	30.1%
Jabref	52.1%	50.5%
JEdit	45.1%	44.4%
SoapUI	51.2%	48.3%

Table VI
OTHER APIs

Project	**RR**
MyTourbook	67.8%
Rodin	49.4%
RssOwl	57.5%

successfully used to recommend methods for APIs other than the Java API.

RQ4: Comparison to Rascal. Table VII shows the results of the comparison to Rascal regarding its own applicability. Column *Appl* shows in how many cases each of the approaches can make recommendations. Rascal is applicable in 96.1% to 98.4% of the cases, whereas our approach can be applied in 68.9% to 89.1% of the cases. The columns RR_{ap} show the recommendation rates for the approaches taking only those method calls into account where the approach is applicable. For a better comparison, RR_{gl} shows the "global" recommendation rate, normalized for all method calls, *i. e.*, it is the product of columns *Appl* and RR_{ap}. Globally, the recommendation rate for Rascal ranges between 24.4% and 32.0%, while our approach recommends the correct method in 33.1% to 43.1% of the cases.

Table VIII shows the recommendation rates for the approaches in cases where both of them are applicable. For these calls, Rascal showed recommendation rates ranging from 19.1% to 33.4%. Our approach was able to recommend the correct method in 41.2% to 51.7% of the cases. The recommendation rate of the ideal hybrid approach is shown in column *Hybrid*. It ranges from 47.5% to 58.8%, which is an increase of 5.5% to 12.4% compared to our approach.

RQ5: Confidence level. The results of the confidence level analysis for the 6 Java applications are shown in Figure 3. For both Rascal and our approach the average recommendation rate relation is monotonically increasing with the confidence interval.

Table VII
COMPARISON TO RASCAL (OWN APPLICABILITY)

Project	**Rascal**			**Our approach**		
	$Appl$	RR_{ap}	RR_{gl}	$Appl$	RR_{ap}	RR_{gl}
DrJava	98.0%	30.5%	29.9%	79.9%	44.3%	35.4%
Freemind	97.2%	25.1%	24.4%	76.9%	43.1%	33.1%
HSQLDB	98.4%	32.5%	32.0%	89.1%	41.6%	37.1%
Jabref	97.8%	26.7%	26.1%	82.8%	52.1%	43.1%
JEdit	97.4%	25.3%	24.6%	73.5%	45.1%	33.1%
SoapUI	96.1%	26.9%	25.9%	68.9%	51.2%	34.8%

Table VIII
COMPARISON TO RASCAL (SHARED APPLICABILITY)

Project	**Appl**	**RR**		
	Shared	Rascal	Our appr.	Hybrid
DrJava	78.7%	31.9%	43.7%	56.1%
Freemind	75.2%	19.1%	42.0%	47.5%
HSQLDB	88.2%	33.4%	41.2%	52.1%
Jabref	81.9%	25.0%	51.7%	58.8%
JEdit	71.8%	26.7%	45.2%	54.0%
SoapUI	66.7%	24.7%	50.2%	57.4%

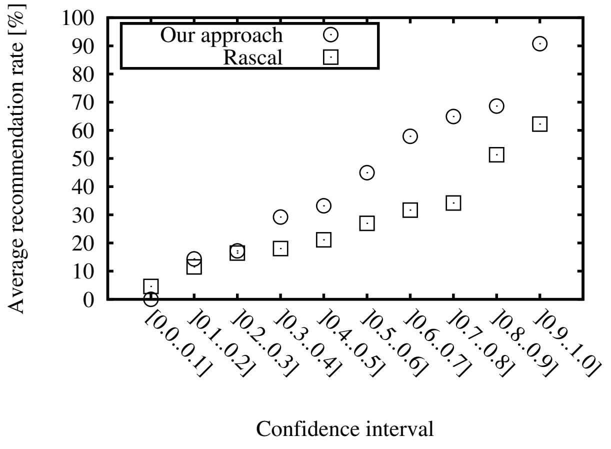

Figure 3. Average recommendation rate vs. confidence

VIII. DISCUSSION

According to the results, our approach includes the correct recommendation in the returned set of 5 methods in about every second case. This is a significant improvement over a naive approach that always returns the 5 most frequent methods, which on average succeeds only in 1 out of 9 cases. A central parameter of our approach is the lookback. As expected, taking too many identifiers into account decreases the recommendation rate, as potentially unrelated identifiers are considered. Taking too few identifiers also has a negative effect, since the descriptive power of the context is reduced. Consequently, there seems to be an optimum value in between, which we determined with our experiments.

In RQ2, we investigated the effect of excluding well-known methods both from mining and the recommendation. As expected, the recommendation rates drop, since well-known methods are used frequently in the source code and are therefore easier to recommend. For the majority of the projects (4 of 6), the decrease is only moderate (below 3%). For two of the projects, we had a significant, although not threatening, reduction in the recommendation rate.

Our results are consistently good for both the Java and the Eclipse API, indicating that the approach is not limited to a specific API. The slightly better results for the Eclipse API could be caused by having a more specific purpose than the Java API. However, to really answer this question, further experiments with other APIs are required.

When comparing our results to method usage-based approaches—here represented by Rascal—we find our approach to return the correct suggestion in 5.5% to 17.0% more of the cases. The reduced recommendation rates of Rascal compared to the numbers reported in [11] and Section V are likely to be caused by the different APIs used. The original experiments were limited to the Swing API, which is a subset of the Java API used in our setup. As the number of possible methods increases, the recommendation rate is expected to drop. Additionally, in the evaluation from [11], the class for which the recommendations are retrieved is part of the training set, while in our experimental setup, the files used for index creation and recommendation are disjunct. While we consider our setup more realistic, it can have an impact on the recommendation rate.

Given that both our approach and Rascal are applicable in slightly different contexts and recommendation rates of about 50% still allow further improvement, a combined approach using both method usage and identifier contexts seems feasible. The results of RQ4 suggest, that such a hybrid approach could add another 5.5% to 12.4% to the recommendation rates of our approach. The question of which results to use if both approaches are applicable for a context is partially answered by RQ5. Our results show that the confidence level we suggested is a good predictor for the expected recommendation rate of both algorithms. Based on this, a recommendation system could calculate recommendations with both algorithms and present the one with the higher confidence level to the user. However, how good exactly this approach would be compared to the ideal hybrid approach is an open question for further research.

The confidence level determined in RQ5 opens another interesting application. A proactive recommendation system could be configured to actively suggest recommendations if the confidence level is sufficiently high, thus not interrupting programming if the results are unlikely to help. To evaluate and assess such a setting would require more details on the interaction of the recommendation system with the user. Additionally, it would have to be evaluated in an experiment with several subjects, which is planned for future work but is beyond the scope of this paper.

Regarding the application in an interactive development environment, we measured times for building and querying the index for each study object. Index construction time was between 30s and 231s. Typically, the construction of the index is performed only once in a preparation step and thus its duration is less important than the query time. The average index query took between 5ms and 26ms, which is sufficient for an interactive setup.

IX. Threats to Validity

Internal validity. The choice of returning 5 methods in the recommendation set is arbitrary and affects the results. However, returning 5 methods seemed to be a suitable compromise from a programmer's perspective and is the same value as used in [11]. Additionally, as all numbers reported are based on 5 methods in the recommendation set, the numbers are comparable to each other.

The context for recommending methods also includes the identifiers in the line of the method call up to the position of the method call. Cases in which this line contains the variable declaration for a very specific return type limit the choice of possible methods and thus increase the probability of a correct recommendation set. However, a user might not know the return type of an appropriate method in advance.

Our evaluation is based on counting how often the method actually used in the code is contained in the recommendation set. This might be different from the recommendation rate actually perceived by a user. However, we argue that this is a one-sided error, as the method actually used in the code should be correct in any case, while a user might even find other methods in the recommendation set useful, even if they were not used in the implementation. Thus, the recommendation rates found in a study with subjects would be expected to be higher rather than lower.

External validity. The results of our experiments might be biased by the choice of study objects. We tried to mitigate this threat by choosing applications from different application domains. Further questions are also how the results transfer to commercial systems (rather than open source) and other programming languages besides Java. While both are interesting questions for further research, we did not try to answer them in this paper.

Our comparison to Rascal could be invalidated by an incorrect reimplementation. However, as explained in Section V, we could reproduce the results of [11] with our implementation, which makes us confident that our implementation of Rascal resembles the published algorithm near enough for a valid comparison.

X. Conclusion and Future Work

In this paper, we described an API method recommendation algorithm based on identifier context. We experimentally found optimal settings for the algorithm's variation points. With these settings, we obtain recommendation rates between 41.6% and 67.8% if the context allows application. Compared to method usage-based approaches, our algorithm is correct in 5.5% to 17.0% more of the cases. Finally, we showed how a confidence level can be used to predict the expected precision of both algorithms, opening the path to a hybrid approach and a more goal-oriented interaction with a user.

A promising next step is the implementation and evaluation of a hybrid recommendation approach that would be expected to have better recommendation rates than both the method usage and the identifier based approach. Additionally, it would be interesting to perform a practical evaluation of the recommendation system in an industrial setting with developers to better understand the impact of recommendation systems on their productivity.

Acknowledgments

We thank the Google Research Awards Program for supporting our research. Moreover, we are grateful to Daniel Ratiu, Benedikt Hauptmann, and Andreas Vogelsang for helpful comments on the paper.

References

[1] C. Krueger, "Software reuse," *ACM Computing Surveys*, vol. 24, no. 2, pp. 131–183, 1992.

[2] W. Lim, "Effects of reuse on quality, productivity, and economics," *IEEE Software*, vol. 11, no. 5, pp. 23–30, 2002.

[3] A. Mili, R. Mili, and R. Mittermeir, "A survey of software reuse libraries," *Annals of Software Engineering*, vol. 5, no. 1, pp. 349–414, 1998.

[4] O. Hummel and C. Atkinson, "Using the Web as a Reuse Repository," *Reuse of Off-the-Shelf Components*, pp. 298–311, 2006.

[5] W. Frakes and K. Kang, "Software reuse research: Status and future," *IEEE Transactions on Software Engineering*, vol. 31, no. 7, pp. 529–536, 2005.

[6] L. Heinemann, F. Deissenboeck, M. Gleirscher, B. Hummel, and M. Irlbeck, "On the Extent and Nature of Software Reuse in Open Source Java Projects," in *ICSR'11*, 2011.

[7] S. Chatterjee, S. Juvekar, and K. Sen, "SNIFF: A Search Engine for Java Using Free-Form Queries," in *FASE'09*, 2009.

[8] D. Eisenberg, J. Stylos, and B. Myers, "Apatite: A new interface for exploring apis," in *CHI'10*. ACM, 2010.

[9] E. Hill, L. Pollock, and K. Vijay-Shanker, "Automatically capturing source code context of nl-queries for software maintenance and reuse," in *ICSE'09*, 2009.

[10] S. Monsell, "Task switching," *Elsevier TRENDS in Cognitive Sciences*, vol. 7, no. 3, pp. 134–140, 2003.

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

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

[13] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman, "Jungloid mining: helping to navigate the API jungle," *ACM SIGPLAN Notices*, vol. 40, no. 6, pp. 48–61, 2005.

[14] L. Heinemann and B. Hummel, "Recommending API Methods Based on Identifier Contexts," in *SUITE'11*, 2011.

[15] F. Deissenboeck and M. Pizka, "Concise and consistent naming," *Software Quality Journal*, vol. 14, no. 3, pp. 261–282, 2006.

[16] A. Zaremski and J. Wing, "Signature matching: a tool for using software libraries," *ACM Transactions on Software Engineering and Methodology*, vol. 4, no. 2, pp. 146–170, 1995.

[17] B. Fischer and G. Snelting, "Reuse by contract," in *FoCBS'97*, 1997.

[18] S. Reiss, "Semantics-based code search," in *ICSE'09*, 2009.

[19] G. Furnas, T. Landauer, L. Gomez, and S. Dumais, "The vocabulary problem in human-system communication," *Communications of the ACM*, vol. 30, no. 11, pp. 964–971, 1987.

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

[21] S. Bajracharya, J. Ossher, and C. Lopes, "Leveraging usage similarity for effective retrieval of examples in code repositories," in *FSE'10*, 2010.

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

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

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

[25] M. Tsunoda, T. Kakimoto, N. Ohsugi, A. Monden, and K. Matsumoto, "Javawock: A Java Class Recommender System Based on Collaborative Filtering," in *SEKE'05*, 2005.

[26] O. Hummel, W. Janjic, and C. Atkinson, "Code Conjurer: Pulling reusable software out of thin air," *IEEE Software*, vol. 25, no. 5, pp. 45–52, 2008.

[27] Sun Microsystems. (2011, Apr.) Code Conventions for the Java Programming Language. [Online]. Available: http://www.oracle.com/technetwork/java/codeconv-138413.html

[28] S. Chakrabarti, *Mining the Web: discovering knowledge from hypertext data*. Morgan Kaufmann, 2003.

[29] Text Fixer. (2011, Apr.) List of common words. [Online]. Available: http://www.textfixer.com/resources/common-english-words.txt

[30] P. Tan, M. Steinbach, V. Kumar *et al.*, *Introduction to data mining*. Addison Wesley, 2006.

[31] H. Ma, R. Amor, and E. Tempero, "Usage Patterns of the Java Standard API," in *APSEC'06*, 2006.