JOURNAL OF SOFTWARE: EVOLUTION AND PROCESS
J. Softw. Evol. and Proc. 2016; **00**:1–23
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/smr

TMAP: Discovering Relevant API Methods through Text Mining of API Documentation

Rahul Pandita[1], Raoul Praful Jetley[2], Sithu D Sudarsan[2], Tim Menzies[1], and Laurie Williams[1]

[1]*North Carolina State University, Raleigh, NC, USA*
[2] *ABB Corporate Research, Bangalore, India*

SUMMARY

Developers often migrate their applications to support various platform/programming-language application programming interfaces (APIs) to retain existing users and to attract new users. To migrate an application written using one API (source) to another API (target), a developer must know how the methods in the source API map to the methods in the target API. Given that a typical platform or language exposes a large number of API methods, manually discovering API mappings is prohibitively resource-intensive and may be error prone. *The goal of this research is to support software developers in migrating an application from a source API to a target API by automatically discovering relevant method mappings across APIs using text mining on the natural language API method descriptions.* This paper proposes TMAP: **T**ext **M**ining based approach to discover relevant **API** mappings. To evaluate our approach, we used TMAP to discover API mappings for 15 classes across: 1) `Java` and `C#` API; and 2) `Java ME` and `Android` API. We compared the discovered mappings with state-of-the-art source code analysis-based approaches: Rosetta and StaMiner. Our results indicate that TMAP on average found relevant mappings for 56% and 57% more methods compared to the Rosetta and the StaMiner approaches, respectively.
Copyright © 2016 John Wiley & Sons, Ltd.

Received . . .

KEY WORDS: API Documents, Text Mining, API Mapping

1. INTRODUCTION

Developers are increasingly releasing different versions of their applications to attract new users and to retain existing users across different platforms. For example, a typical mobile software developer often releases his/her applications on all the popular mobile platforms, such as Android, iOS, and Windows, which often involves rewriting applications in different languages. For instance, Java is the preferred language for implementing Android applications and Swift for iOS applications. In the context of desktop software, many well-known projects, such as JUnit and Hibernate, provide multiple versions in different programming languages to attract the developer community to use these libraries across those languages.

Existing tools, such as Java2CSTranslator [16], assist developers by automating the process of software migration. However, such tools require programmers to manually input how methods in a source language's Application Programming Interfaces (API) maps to the methods of the target language's API. Given that a typical language (or platform) exposes a large number of API methods

*Correspondence to: North Carolina State University, 890 Oval Drive Campus Box 8206, 3228 EB-II, Raleigh, NC 27695-8206. Email: rpandit@ncsu.edu

Copyright © 2016 John Wiley & Sons, Ltd.

for developers to reuse, manually discovering these mappings is prohibitively resource intensive and may be error prone.

Existing program-analysis-based approaches address the problem of finding method mappings between APIs using static [40] and dynamic [12] analysis. Recently, Nguyen et al. [24] further proposed to apply statistical language translation techniques to achieve language migration by mining large corpora of open source software repositories. However, these approaches require as an input manually-ported software or at least functionally-similar software artifacts (source-code or binaries) across source and target APIs. Since static analysis approaches [24, 40] leverage source code analysis, the accuracy of such approaches is dependent on the quality of the code under consideration. Likewise, the accuracy of dynamic approaches [12] is dependent on the quality and completeness of test inputs to exercise the API behavior comprehensively. Furthermore, such program analysis approaches are often sensitive to the nuances of the programming language of the source code under analysis, thus require significant effort for accommodating new programming languages.

The goal of this research is to support software developers in migrating an application from a source API to a target API by automatically discovering relevant method mappings across APIs using text mining on the natural language API method descriptions.

We propose to use the natural language API method descriptions to discover the method mappings across APIs. We hypothesize: *since the API documents are targeted towards developers, there may be an overlap in the language used to describe similar concepts that can be leveraged.* In general, API documentation provides developers with useful information about class/interface hierarchies within the software. Additionally, API documents also provide information about how to use a particular method within a class by means of method descriptions. A method description typically outlines specifications in terms of the expectations of the method arguments and functionality of method in general.

This paper presents TMAP: An approach that leverages the natural language method descriptions to discover the likely method mapping between APIs. TMAP stands for ***T**ext **M**ining based approach to discover likely **API** method mappings*. TMAP accepts as input the API documents of the source and target API. In particular, TMAP proposes to create a vector space model [21, 33] of the target API method descriptions. TMAP then queries the vector space model of the target API using automatically-generated queries from the source API method descriptions. TMAP automates the query generation in source API using the concepts from text mining, such as emphasizing certain keywords over others and querying multiple facets of API documents, such as class descriptions, package names, and method descriptions. The output of TMAP is a ranked list of methods of the target API that are candidates for the mapping of the method from the source API that was used to generate the query. Since TMAP analyzes API documents in natural language, the proposed approach is reusable, independent of the programming language of the library.

However, automatic query generation is challenging. In particular, given that a typical API exposes a large number of methods, a large search space may result in noise in the search results. Consider for example, Android API level 23 exposes 4,404 [†] classes and Java 8 exposes 4,240 [‡] classes. Furthermore, each class exposes several methods resulting in a large search space for each query and low search effectiveness. TMAP addresses this challenge by leveraging feedback in the form of confirmed mappings to prune the search space using topic modeling [4, 29]. Thus, given a set of confirmed mappings, TMAP improves the recommendations for future queries by pruning irrelevant methods from search results.

We pose the following research questions:

- **RQ1**: Does TMAP discover additional method mappings in comparison to existing program-analysis based approaches?
- **RQ2**: What is the improvement in pruning irrelevant mappings from the TMAP output by leveraging feedback?

[†] http://developer.android.com/reference/classes.html
[‡] https://docs.oracle.com/javase/8/docs/api/allclasses-noframe.html

Copyright © 2016 John Wiley & Sons, Ltd.
Prepared using smrauth.cls

DOI: 10.1002/smr

- **RQ3**: What is the overlap of the method mappings discovered by TMAP in comparison with the mappings discovered by existing program-analysis based approaches?
- **RQ4**: Does TMAP discover relevant methods using free-form queries instead of automatically generated queries?

To answer our questions, we apply TMAP to discover likely API mappings for 15 classes across: 1) `Java` and `C#` API; and 2) `Java ME` [§] and `Android` API. We also compare the discovered mappings with two state-of-the-art program-analysis based approaches: Rosetta [12] and StaMiner [24].

This paper makes the following major contributions:

- A text mining (queries on vector space model) based approach that effectively discovers mapping between source and target API.
- A topic modeling-based feedback approach for reducing the search space of queries to prune irrelevant suggestions in discovering mapping between source and target API.
- A prototype implementation of our approach based on extending the Apache Lucene [20] and Mallet [22] libraries. An open source implementation of our prototype can be found on our project website [30].
- An evaluation of our approach on 5 classes in `Java ME` to `Android` API and 10 Classes `Java` to `C#` API. The evaluation results and artifacts are publicly available on the project website.

This paper is a revised, expanded version of a paper (Pandita et al. [26]) presented at the 15th IEEE working conference on Source Code Analysis and Maintenance (SCAM 2015). Specifically, this paper builds on the previous version by proposing and evaluating a new technique to leverage feedback for pruning the irrelevant TMAP suggestions. Furthermore, this paper also evaluates the effectiveness of text-mining in general for API related information retrieval tasks.

The rest of the paper is organized as follows. Section 2 presents the background on Text Mining. Section 3 presents a real world example that motivates our approach. Section 4 presents the TMAP approach. Section 5 presents our evaluation of TMAP. Section 6 presents a brief discussion and future work. Section 7 discusses the related work. Finally, Section 8 concludes the paper.

2. BACKGROUND

Text mining is a broad research area, including but not limited to the techniques facilitating retrieval and manipulation of useful information from a large corpus to text. As opposed to traditional data mining, text mining analyzes free-form text distributed across documents (rather than data localized and maintained within a database). To analyze this data, text mining uses concepts from traditional data analytics, natural language processing, and data modeling domains. We next introduce the text mining concepts that have been used in the presented approach.

Indexing [11, 21]: Indexing is the process of extracting text data from source documents and storing them in well-defined indices. During the indexing process, a document (a sequence of text) is divided into its constituent units, known as tokens or terms, based on a well-defined criterion. A term is typically an individually identifiable unit of the document (such as a word) and relates to the individual terms stored in an index. Once the terms within each document are identified, they are added to the index, with the corresponding link to the document and associated term frequencies. Term frequency is the simple count of the occurrence of a term in a document. An optional pre-processing step further assists with indexing, such as removing stop-words.

Among various indexing strategies, the use of *inverted file indexing* [11] is well suited for large document collections. In the simplest form, an inverted file index provides a mapping of terms, such as words, to their locations in a text document. A document can be thought of as a collection of m terms. Typically a document is made up of a sequence of n unique terms such that $n <= m$. The number n is usually far less than m as most of the terms are repeated while forming a document. For

[§] Java Platform Micro Edition

Copyright © 2016 John Wiley & Sons, Ltd.
Prepared using smrauth.cls

DOI: 10.1002/smr

instance, the term "the" is repeated many times in this paper. The set of unique terms within an index forms the "Term List" v of the index. If a pointer (such as a numeric location) is associated with each term in v to the location of that term in a text document, the resultant data structure is a form of an inverted file index. As the document collection grows, the number of documents matching a term in the index becomes more sparser.

The index is oftentimes annotated with information regarding the frequency of occurrence of each term in the document. The representation of a document as a vector of frequencies of terms is referred to as *vector space model* [11, 33].

TF-IDF [21]: Term -frequency inverse-document-frequency (`tf-idf`) is a numerical statistic that is intended to capture the importance of a term to a document in a corpus. Often used as a weighting metric in information retrieval and text mining, the `tf-idf` weight increases proportionally to frequency of the occurrence of a term in a document; however, the weight is also offset by the count of documents that contain the term.

Cosine Similarity [33]: In mathematics, cosine similarity is a numerical statistic to measure the similarity between two vectors. Cosine similarity is defined as the dot product of magnitude of two vectors. In the context of text mining, the cosine similarity is used to capture the similarity between two documents represented as term frequency vectors.

Topic Modeling: Latent Dirichlet Allocation (LDA) [4,29] is an Information Retrieval (IR) model that clusters the documents in a corpus as a function of the probability of occurrences of topics in those documents. A topic itself can be thought of as a collection of terms that can be collectively interpreted as connected. For instance, a topic with the following terms "*draw*", "*graphics*", and "*paint*" can be interpreted as *drawing*-related topic. LDA accepts as input a corpus of documents and the following parameters as described in [29]:

1. k, the total number of topics to be extracted from a corpus of documents.
2. n, the total number of Gibbs iterations, where a single iteration involves a Gibbs sampler sampling a topic for each term occurring in the corpus.
3. α, this parameter affects the distribution of topics across documents. A high α means that each document is likely to contain a mixture of most of the topics. Conversely, a low α means each document is likely to contain mixture of fewer (or one) topics.
4. β, this parameter affects the distribution of terms in each topic. A high β means that each topic is likely to contain a uniform distribution of terms. Conversely, a low β means terms are not uniformly distributed across topics.

Given a corpus (C) of m documents as input, LDA first creates a dictionary (D), which is a list of all the unique terms (n) occurring in the corpus. After creating the dictionary, LDA constructs a weighted-term-to-document matrix $M(mXn)$. Each row in M corresponds to a document in the corpus. Each cell in the row represents weight of each term in D for the document represented by the row. For instance, $M_{[i,j]}$ represents the weight of j^{th} term in D for the i^{th} document in C. In the simplest implementation `tf-idf` is used to calculate the weight of a term in a document.

Next, given a number of topics (k) as an input parameter, LDA transforms the weighed term-to-document matrix M to topic-to-document matrix (θ) of size $\theta(mXk)$. This transformation is achieved by identifying latent variables (topics) in the documents [4]. Each row in θ corresponds to a document in the corpus. Each cell in the row represents the weight of each *topic* in D for the document represented by the row. For instance, $\theta_{[i,j]}$ represents the probability of j^{th} topic occurring in the i^{th} document in C. Since $k << n$, LDA can be thought of as a mapping of the documents from the term space n to the topic space k [29].

3. EXAMPLE : SHORTCOMINGS OF KEYWORD-BASED SEARCH

We next present an example to motivate our work and list the considerations for applying text mining techniques on API documents. The presented example is from the `Java ME` and the `Android` API. Both Java ME and Android use Java as the language of implementation and are targeted

Copyright © 2016 John Wiley & Sons, Ltd.
Prepared using smrauth.cls

DOI: 10.1002/smr

javax.microedition.lcdui Class Graphics drawString

```
public void drawString(String str,int x,int y,int anchor)
```

Draws the specified String using the current font and color. The x,y position is the position of the anchor point. See anchor points.

Parameters

str - the String to be drawn
x - the x coordinate of the anchor point
y - the y coordinate of the anchor point
anchor - the anchor point for positioning the text

Throws

NullPointerException - if str is null
IllegalArgumentException - if anchor is not a legal value

See Also

`drawChars(char[], int, int, int, int, int)`

Figure 1. `drawString` API method description of `Graphics` class in the Java ME API

android.graphics Class Canvas drawText

```
public void drawText (String text, float x, float y, Paint paint)
```

Draw the text, with origin at (x,y), using the specified paint. The origin is interpreted based on the Align setting in the paint.

Parameters

text - The text to be drawn
x - The x-coordinate of the origin of the text being drawn
y - The y-coordinate of the origin of the text being drawn
paint - The paint used for the text (e.g. color, size, style)

Figure 2. `drawText` API method description of `Canvas` class in the Android API

towards hand-held devices. Figure 1 shows the API method description of `drawString` method from `javax.microedition.lcdui.Graphics` class in Java ME API. Figure 2 shows the API method description of method `drawText` method from `android.graphics.Canvas` class in Android API.

Notice the overlap in the language of these two methods. A human developer can effortlessly (or with a little overhead) conclude that the two methods offer similar functionality. However, Android API has more than 23,000 public methods. Manually going through each method description to find similar methods is prohibitively time consuming, supporting the need for automation. A naive solution to automate the task is to perform keyword-based search.

To demonstrate the difficulties faced by keyword-based search in the above example, we searched the Android API description with the keywords listed in Table I using the Apache Lucene [20] framework. Lucene is a high-performance, full-featured text search engine library written entirely in Java. The column "Query" describes the keywords we used to perform keyword-based search, the column "Hits" lists the number of matches found, and the column "Top-10" lists the rank of the first relevant method in top-ten results. For instance, when we searched for the class name "Graphics" in the Android API, we did not get any results. We also did not get any results for when we searched for method name using keywords "drawString".

When we searched the Android API using the words in the method signature as keywords, we got 23,547 results (almost all of the methods in Android API). The high number of results is because of the confounding effects of keywords, such as "public". Most of the method signatures

Copyright © 2016 John Wiley & Sons, Ltd.
Prepared using smrauth.cls

DOI: 10.1002/smr

Table I. Query Results

#	Query	Hits	Top-10
1	Class Name: "Graphics"	0	-
2	Method Name: "drawString"	0	-
3	Method Signature	23547	-
4	Method Description: (Complete)	16820	-
5	Method Description: (summary sentences)	94230	-
6	Combination	1479	**3**

'-'=No Match in Top-10 results.

have the keyword "public". Although the ranking mechanisms in Lucene did rearrange the results moving methods with most keyword matches first, we did not find a relevant method in top ten results. Likewise, the words in method descriptions as a whole or in parts also did not yield better results. In the example, a combination of various attributes, such as method name split in camel case notation "draw String", keywords from both class and method description resulted in the Android API equivalent method `drawText` shown in Figure 2 in the top ten results.

The previous example demonstrates the difficulties faced by simple keyword-based searches, and text mining in general, to discover likely method mappings. The TMAP approach addresses the following difficulties in applying text mining approaches on natural language API method descriptions:

1. ***Confounding effects***. Certain method names have a confounding effect. For instance, `toString()`, `get()`, `set()` are too generic and tend to have similar descriptions across many method definitions. These generic methods often cause interference with the output of text mining based approaches. The challenge is to automatically identify such method names to de-emphasize their importance in a query.
2. ***Weights***. Not all terms in a method descriptions are equally important keywords. For instance, the term "zip" in the sentence "opens a zip file" is more important than the terms "opens" and "files" as the term emphasizes on the specific type of file. The challenge is to automatically identify the importance of a term.
3. ***Structure***. API documents are not flat, contiguous text blobs. They have well defined structure, that is often shared by method descriptions. Ignoring the structure may cause ineffective queries to negatively affect the results. The challenge is to effectively aggregate the results of querying individual API document elements (such as class descriptions, class names, method names, and method descriptions).

4. TMAP

We next present our approach for discovering likely mappings of API methods across APIs. Figure 3 provides an overview of the TMAP approach. The TMAP approach consists of three major components: 1) an Indexer; 2) Query Builder; and 3) Searcher components.

The Indexer accepts the API documents of the target API and creates indices (a vector space model) of these documents by extracting intermediate contents from the method descriptions. The Query Builder accepts the API documents of the source API and creates queries to be executed on the indexes (a vector space model). In addition, the Query Builder component also accepts feedback in the form of confirmed mappings to further refine the query. Finally, the Searcher component executes the queries on the indices and generates a ranked list of the API methods from target API documents as mapping results to be presented to the developers for confirmation.

We next describe each component in detail.

Copyright © 2016 John Wiley & Sons, Ltd.
Prepared using smrauth.cls

DOI: 10.1002/smr

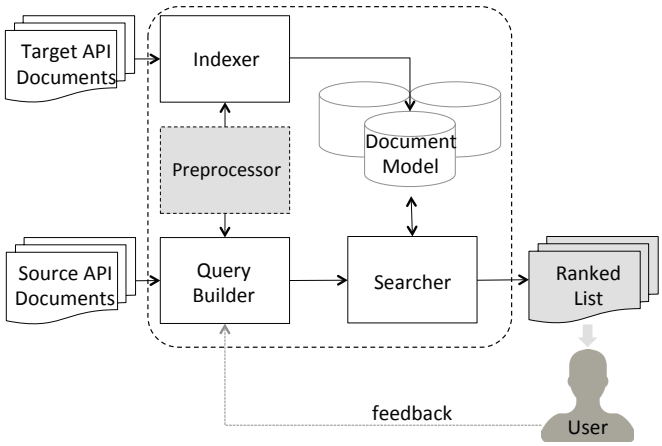

Figure 3. Overview of TMAP approach

4.1. Indexer

This component accepts the API method descriptions of the target API and creates indices (a vector space model) of these documents. Specifically, Indexer extracts the following fields from the API method descriptions:

F1) Type Name: The name of enclosing class or interface of the method. For the method description of `drawString` shown in Figure 1, Indexer extracts the type Name as "Graphics".

F2) Package Name: The package name of the enclosing type. For the method description shown in Figure 1, Indexer extracts the package name as "javax.microedition.lcdui".

F3) Method Name: The name of the method. For the method description shown in Figure 1, Indexer extracts the method name as "drawString".

F4) Class Description: The description of the enclosing type. Class description is not shown in Figure 1 for space considerations.

F5) Method Description: The description of the method. For the method description shown in Figure 1, Indexer extracts the method description as all the text except for the method declaration in Line 1.

This step is required to extract the desired descriptive text from the API method descriptions. Getting structured descriptions facilitates searching on individual categories or facets. This step allows TMAP to deal with the *structure* issue, as presented in Section 3. Different API documents may have different styles of presenting information to developers. Such stylistic differences may also include the difference in the level of the detail presented to developers. TMAP relies only on basic fields that are generally available for API methods across different presentation styles.

After extracting the desired information, the extracted text is further preprocessed. The preprocessing steps are required to make the text amenable to text mining techniques that are used in the subsequent phases of the TMAP approach. In particular, TMAP performs the following basic prepossessing steps:

P1) Presentation Elements: A typical API method description is often interleaved with presentation elements for improved readability. For instance, `JavaDoc` provides a list of identifiers such as `@Code` and `@link`. These elements are automatically translated into presentation markup,

Copyright © 2016 John Wiley & Sons, Ltd.
Prepared using smrauth.cls

DOI: 10.1002/smr

such as links and fonts. Although such elements are part of the description text, they often cause hindrance for the text mining techniques to compute relevance based on the query. Therefore, this preprocessing step cleans the method descriptions to remove such elements. We use a static list of presentation elements to achieve cleaning in this step with relatively high accuracy.

P2) Split Package Notation: In method descriptions, the "." character is used as a separator character for package names like "javax.microedition.lcdui". We use regular expressions to split the package name into constituent words to facilitate search on individual words in the package name. For example, "javax.microedition.lcdui" is split into "javax microedition lcdui".

P3) Split CamelCase Notation: API method descriptions are often interleaved with programming identifiers, such as class names and method names. Oftentimes these identifiers use CamelCase notation. The CamelCase notation has become the *de facto* mechanism used by programmers for combining phrases into a single word, such that each word in the phrase begins with a capital letter. Previous research [19] demonstrated the benefit of splitting the CamelCase word into its constituent phrase for automated code completion. TMAP splits such identifiers into constituent phrases to facilitate effective searching. TMAP leverages the well-formed structure of CamelCase notations to encode a regular expression to achieve splitting with relatively high accuracy. For example, "drawString" is split into "draw String".

P4) Lowercase: This step involves converting the text description to lowercase. This step is performed to normalize the text for making the keyword match case insensitive, further increasing the range of queries.

P5) Stemming: This steps transforms the words in the description to their base form. Stemming is very effective in extending the range of keyword based queries to match various operational forms of the words. For instance, "has", "have", and "had" are mapped to the stem "ha". We use the default implementation of the Lucene Porter stemmer for preprocessing.

After preprocessing, TMAP creates indices for the API method descriptions. An index is a collection of documents where each document is made up of values organized into well defined fields. TMAP considers every method description as an individual document and uses the following major fields (as previously described): 1) combination of package and class name; 2) class description; 3) method signature; 4) method name; and 5) method description. The values of these fields are the text after preprocessing. TMAP uses a vector space model representation of the documents for each field. Vector space model or term vector model is an algebraic model for representing text documents (and any objects, in general) as vectors of identifiers and their frequency of occurrence. In case of TMAP, each word is considered as a term except the stop words, such as "a", "the", and "and".

4.2. Query Builder

This component accepts the API method descriptions of the source API and creates queries for method descriptions. The queries are further refined by leveraging the feedback provided by the end user in the form of confirmed mappings. The confirmed mappings are used to prune the irrelevant results of the queries. These queries are executed on the target API index to retrieve an ordered list of relevant API methods. In particular, Query Builder uses the same preprocessing steps followed by Indexer (listed in Section 4.1). After extracting the desired descriptive text from the API method descriptions, this component systematically creates search queries to search for different fields in indexes. Keywords for searching in "Type Name", "Type Description", "Method Name", and "Method Description" fields are derived from their equivalents in the extracted descriptive text.

For instance, consider the method description shown in Figure 4 and equivalent query in shown in Figure 5. The keywords for "Type Name" are derived from preprocessing "java.util.Iterator" resulting in "java util iter". Notice the package notation is split into individual words and "Iterator" is further transformed to lower case and its stem "iter". Likewise, keywords for the field "Method Name" are derived by preprocessing "hasNext", which is first split into "has Next" and then transformed using stemming into "ha next" (*ha* being stem of word *has*).

Copyright © 2016 John Wiley & Sons, Ltd.
Prepared using smrauth.cls

DOI: 10.1002/smr

java.util Interface Iterator **hasNext**
`boolean hasNext()`
Returns `true` if the iteration has more elements. (In other words, returns true if `next()` would return an element rather than throwing an exception.)
Returns:
`true` if the iteration has more elements

Figure 4. API Method Description of `hasNext` method in `Iterator Interface` from the Java API

Type Name: `java util iter`
Type Description: `iter over collect iter take enumer java collect framework`
Method Name: `ha next`
Method Description: `return true iter ha more element other word return true next would return element rather than throw except`

Figure 5. Query based on API method description of `hasNext` method in `Iterator Interface` from the Java API

To generate keywords for querying the "Type Description" field we consider the following heuristic: *Heuristic_H1: the first paragraph or the first five sentences of the type description (whichever is shorter) provides reasonable keywords for searching equivalent class in target API.*

Likewise, to generate the keywords for querying the "Method Description" field, we consider the following heuristic: *Heuristic_H2: the first paragraph or the first two sentences of the method description (whichever is shorter) provides reasonable keywords for searching equivalent methods in target API.*

TMAP uses these heuristics to improve the performance of the searching infrastructure that tends to be inversely proportional to the complexity and length of the query. Using all the descriptive text as keywords often results in a very verbose query. As the number of keywords in a query increases, the effectiveness of the query decreases. A large number of keywords have higher probability of matching a large number of documents in comparison to a query with fewer keywords. In contrast, we observed that the document writers tend to describe the general overview of class and method descriptions in the first few sentences, followed by implementation and design specific details. We thus focused on the words in these overview sentences to create queries, instead of using the entire descriptive text.

Weights for terms. As mentioned in Section 3, all terms in a method description are not equally important keywords. TMAP further enhances the query by quantifying the importance of a term in the method description and using that as the weight of the corresponding keywords in the query. In particular, we propose to use tf-idf score [21] as a metric to quantify the importance of a term. For each term in the method descriptions TMAP calculates the number of times the term occurs in that method description as $freq_{mtd}$. TMAP also calculates the maximum frequency of any term in the document as $freq_{MAX}$. TMAP then calculates the number of documents in the corpus that contains the term as $freq_{doc}$. TMAP finally calculates the tf-idf score of the term (as listed [21]) as:

$$\texttt{tf-idf} = (0.5 + \frac{0.5\ X\ freq_{mtd}}{freq_{MAX}}) * log(1 + \frac{total_{mtd}}{freq_{doc}})$$

The calculated tf-idf values of terms are normalized to a range of 0 to 1 (both 0 and 1 inclusive) for each document. The normalized tf-idf score of the top-k terms is then used as the weights for the corresponding keywords occurring in the query.

Copyright © 2016 John Wiley & Sons, Ltd.
Prepared using smrauth.cls

DOI: 10.1002/smr

For the API method description shown in Figure 4, TMAP calculates "iter", "ha" (Stem of "has"), and "element" as most the important terms with normalized tf-idf scores of 1.0, 1.0, and 0.6 respectively. We augment the query shown in Figure 5 with the computed weights for the keywords respectively.

4.2.1. Feedback The Query Builder component also accepts feedback in the form of confirmed mappings to further refine the results of a query. A confirmed mapping is a method mapping that has been accepted as a valid mapping by a user. This goal of having this component in TMAP is to prune the irrelevant results of a given query. Without the proposed feedback, the search space for the queries is all the public methods of the target API. Given that a typical API exposes a large number of methods, a large search space may result in noise in the search results in the form of irrelevant results. Consider for example, Android API level 23 exposes 4,404 [¶] classes and Java 8 exposes 4,240 [‖] classes. Furthermore, each class exposes several methods resulting in a large search space for each query, in turn resulting in a low search effectiveness.

A well designed API is typically modular [5]. We seek to leverage the modularity in the API to prune the irrelevant query results. API methods are modular as they are arranged according to the functionality provided by the methods. This arrangement manifests as method groups appearing in a class and class groups appearing in a package. Our key intuition is that mappings of methods localized in a source API will also be localized in the target API. This intuition is based on the statistical model translation theory [6] that given a context, a set of closely related words and the set of their translations are likely to occur frequently together in different languages.

Data: TopicModel T, List Doc_{src}, List Doc_{target}, Integer k
Result: Map< $Topic, List_{Documents}$ > TopicMap
List Doc_{master} = combineLists (Doc_{src}, Doc_{target});
initialize TopicMap;
for *all topics in* T **do**
  get current topic t;
  List$_{Documents}$ L = query Doc_{master} for Topic t ;
  TopicMap.put(t, top-k(L));
end
return TopicMap;

Algorithm 1: Create TopicMap

TMAP operationalizes our intuition using topic modelling [4, 29]. Topic modeling has been demonstrated to be useful in many information-retrieval-related software engineering tasks [3, 29]. In particular, TMAP uses LDA as the choice of topic modeling implementation. LDA creates a probabilistic statistical model that estimates distributions of latent topics from textual documents. TMAP considers each package as an individual document. In particular, we concatenate the package description with the descriptions of the classes and methods occurring within the package to form the document.

TMAP first creates a topic model T of source API documents with m topics. TMAP next combines the source and target API documents to query the model (T) and generate a topic map that is a ranked list of top-k API documents L per topic m. Algorithm 1 lists the steps to create a topic map.

Next, for a given source API document D_s, TMAP discards every L where D_s does not appear to get a subset of lists $\{L\}'$. TMAP next assigns weights to each target API document D_t in each list in $\{L\}'$ as a logarithmic function of their rank. In particular, if D_t appears at rank r in list of size k, the weight assigned is $log(1 + (k - r))$. TMAP next combines the target API documents into a single ranked list L_t by accumulating weights for each unique API document appearing in

[¶] http://developer.android.com/reference/classes.html
[‖] https://docs.oracle.com/javase/8/docs/api/allclasses-noframe.html

Copyright © 2016 John Wiley & Sons, Ltd.
Prepared using smrauth.cls

DOI: 10.1002/smr

Data: TopicMap M, Document D_s, List Doc_{target}, Integer k
Result: List$_{Documents}$ rankedList
initialize weight of each document in Doc_{target} to 0;
for *all Topics* T *in* M **do**
  get List$_{Documents}$ (L) for current topic T;
  if L *contains* D_s **then**
    for *all Documents* D *in* L **do**
      if D *is not in* Doc_{target} **then**
        remove D from L;
      end
    end
    update L for topic T in M;
  else
    remove T and L from M;
  end
end
for *all Topics* T *in* M **do**
  get List$_{Documents}$ (L) for current topic T;
  for *all Documents* D *in* L **do**
    δ = log(1 + (size of List$_{Documents}$- rank of D in List$_{Documents}$);
    increment weight of D in Doc_{target} by δ;
  end
end
Sort Doc_{target} by decreasing order of weights;
return top-k Documents in Doc_{target} as rankedList;

Algorithm 2: Get Ranked List of Target Documents for a Source Document

$\{L\}'$. Algorithm 2 lists the steps to get L_t from the topic map generated in Algorithm 1. Finally, TMAP uses the top-ten resulting documents in L_t to prune the irrelevant results of the query to those packages.

However, LDA requires setting up a lot of parameters for effectively generating a topic model. For instance, a typical implementation of LDA requires the user to set the following four parameters : 1) Number of topics; 2) Number of iterations; 3) Dirichlet distribution parameters α; and 4) Dirichlet distribution parameter β. Finding optimal values for these parameters is an active area of research. TMAP uses the recommendation by Arun et al. [2] of optimizing for the number of topics m by running LDA under different values of m, while maintaining the rest of the parameters as constant. TMAP then use a fitness function to find the optimal value of m.

TMAP uses hyperparameter optimization [22] to optimize values for α and β (α and β are initialized to recommended [22] 1.0 and 0.01 respectively). The parameter "number of iterations" is initialized to 10,000 (large for hyperparameter optimization to be effective). TMAP next creates a set of topic models by running LDA with following values of m: 10, 50, 100, 500, 1,000, 2,500, and 5,000. The following fitness function is then used to select an optimal topic model:

Given a mapping from the feedback; *that methods in source package (P_S) are mapped to methods in target packages ($P_{T_1}...P_{T_n}$)*, all the generated topic models are queried with P_S. The topic model with minimum m containing most of ($P_{T_1}...P_{T_n}$) in top-ten results for similarity described previously is selected as the optimal model for pruning irrelevant results in query output.

We envision that a user may follow an iterative process to leverage feedback. A user may initially use TMAP without the feedback, and validate output for initial queries. He/she will then use the validated output as confirmed mappings to refine the subsequent query results. As an alternative, a user can also leverage the results from existing approaches such as Rosetta [12] and StaMiner [24] as feedback.

Copyright © 2016 John Wiley & Sons, Ltd.
Prepared using smrauth.cls

DOI: 10.1002/smr

4.3. Searcher

The searcher component accepts the query from the Query Builder component and queries the index generated by the Indexer component. The results are then ranked and presented to the end user for review. The searcher is realized as follows. First all the documents that match the keywords and clauses in a query are returned. Then, the returned documents are ranked using the cosine similarity [33] of the terms in the query and the terms in returned documents. In mathematics, cosine similarity is a numerical statistic to measure the similarity between two vectors. In information theory [21], cosine similarity is the standard statistic to rank relevant documents.

4.4. Implementation

We implemented a prototype version of the TMAP approach. First we manually downloaded the HTML version of API documents of the libraries. We then implemented a parser to extract the requisite text from these documents using Jsoup[**], which is a Java library for working with HTML documents. In particular, our prototype implementation parses: 1) Oracle's Javadoc style; 2) Android style documentation; and 3) Microsoft's MSDN documentation.

We next implemented the indexer, query builder, and searcher infrastructure using the Apache Lucene [20] framework. Lucene is a high-performance, full-featured text search engine library written entirely in Java. We used Mallet [22] framework as our choice of topic-modeling library to implement feedback mechanism for the search space pruning. Our prototype implementation and evaluation subjects are publicly available on the project website [30].

5. EVALUATION

We conducted an evaluation to assess the effectiveness of TMAP. In our evaluation, we address following research questions:

- **RQ1**: Does TMAP discover additional method mappings in comparison to existing program-analysis-based approaches?
- **RQ2**: What is the improvement in pruning irrelevant mappings from the TMAP output by leveraging feedback?
- **RQ3**: What is the overlap of the method mappings discovered by TMAP in comparison with the mappings discovered by existing program-analysis based approaches?
- **RQ4**: Does TMAP discover relevant methods using free-form queries instead of automatically generated queries?

All the experiments were performed on a MacBookPro with 2.4 GHz Intel Core i7 processor and 8GB of RAM, running *OS X Yosemite*. The indexing of API Documents took less than a minute [††]. Response time for each query was less than a second.

5.1. Subjects

We evaluated TMAP using snapshots of the publicly available API documents of `Java ME`, `Android`, `Java`, and `C#` downloaded in January 2015. We chose these subjects to enable comparisons with prior case studies of the program analysis-based approaches: Rosetta [12] and StaMiner [24].

Java Platform Micro Edition, or Java ME, is a Java platform designed for embedded systems (such as mobile devices). Target devices range from industrial controls to mobile phones (especially feature phones) and set-top boxes. Android is a Linux-based operating system designed primarily for touchscreen mobile devices, such as smartphones and tablet computers. API documents for both Java ME and Android are publicly available at [15] and [1] respectively.

[**] `http://jsoup.org/`

[††] The reported time is subjected to availability of API documents local machine.

Copyright © 2016 John Wiley & Sons, Ltd.
Prepared using smrauth.cls

DOI: 10.1002/smr

Java and C# are general-purpose programming languages from Oracle and Microsoft, respectively. Java applications are typically compiled to bytecode that run on any Java Virtual Machine (JVM) irrespective of underlying computer architecture. Likewise, C# is compiled into an intermediate representation that run on Microsoft's common language infrastructure. API documents for both Java and C# are publicly available at [17] and [9] respectively.

Specifically, we used the API documents of the following library pairs as subjects for our evaluation:

Java ME (to Android) API: For our evaluation, we considered the methods in the following Java ME `types` as the source API methods to discover mapping methods in Android API: `Alert`, `Canvas`, `Command`, `Graphics`, and `Font` classes in the `javax.microedition.lcdui` package.

The listed types provides methods for supporting graphics related functionality in Java ME. Furthermore, the Rosetta approach by Gokhale et al. [12] reports the mapping for methods in these types along with seven others (twelve types in total) as a part of their evaluation, thus allowing a comparison with dynamic-analysis based approaches. The Rosetta approach requires a user to manually execute functionally-similar applications using source and target API with identical (or near identical) inputs and collect execution traces. Finally, Rosetta analyzes the collected execution traces to infer method mappings. We focused our evaluation on the five listed types, which the first three authors perceived as frequently used types among the twelve types reported by Rosetta. In the future, we plan to evaluate the TMAP approach on all the twelve reported types.

Java (to C#) API: For our evaluation, we considered the methods in the following Java `types` as the source API methods to discover mapping methods in C# API: 1) `File`, `Reader`, and `Writer` in `java.io` package; 2) `Calendar`, `Iterator`, `HashMap`, and `ArrayList` in `java.util` package; and 3) `Connection`, `ResultSet`, and `Statement` classes in `java.sql` package.

The types in `java.io` provides the API methods for accessing and manipulating the file system. The types in `java.util` provides API methods for miscellaneous utilities, such as text manipulation, collections frameworks, and other data structures. Types in `java.sql` provides the API methods for accessing and processing data stored in databases. We selected these particular packages in Java programming language API as Nguyen et al. [24] in their work (StaMiner) for statistical language migration find mappings for the types in these packages. Their mapping results allow a comparison of TMAP with static-analysis based approach. Although, Nguyen et al. [24] report on all the classes in these packages, we focused our analysis on the listed types, which the first three authors perceived as frequently used types in their respective packages.

5.2. Evaluation Setup

We first downloaded the publicly-available API documents from the respective websites of the subject APIs. We then cleaned and extracted the desired fields, as described in the Section 4.1, and indexed the extracted text into the Lucene indexes. We created a separate index for each API type: Java ME, Android, Java, and C#.

For every type (class/interface) under consideration (as listed in Section 5.1), we extracted the publicly-listed methods from API documents. For a given type, we only considered the methods that are listed in the public API. We only considered the methods explicitly declared or overridden by a type and ignore the inherited methods. We then used TMAP to create the queries form the descriptions of the considered methods, as described in Section 4.2. Finally, we executed the formulated queries on the index and collect results. The result of a query execution is a ranked list of methods from the target API that are candidates for mapping. We only considered top-10 results for each query. Previous approaches [7, 12] also only consider the top-10 results suggested by their approach for evaluation.

The top-10 matches found by the TMAP were then manually analyzed/reviewed to determine the effectiveness of the matched results. For a given method in the source API, a match is characterized by a class and a method within that class that is determined to be the corresponding implementation in the target API. The authors next annotated each match as "relevant' ' and/or "exact" based on the following acceptance criteria:

Copyright © 2016 John Wiley & Sons, Ltd.
Prepared using smrauth.cls

DOI: 10.1002/smr

1. **Relevant**: If the target method in the top-10 list can be used to implement the same (or similar) functionality as the source method, we classified the result as relevant.
   OR
   The target method is reported by the previous approaches [12, 24] as a mapping.
2. **Exact**: If the target method is a relevant method and the target method accurately captures the functionality of the source method, and implements the same feature/function, the resultant match is classified as an 'exact' match.
   OR
   The target method is reported by the previous approaches [12, 24] as a mapping.

For example, the `getInt` method in `java.sql.ResultSet` type has an exact match in the C# method `GetInt32` from the `system.data.sqlclient.SqlDataReader` type, since both methods provide the same functionality of extracting the 32-bit signed integer value stored in a specified column. In contrast, the `getClob` method in `java.sql.ResultSet` type does not have an exact corresponding method in C#. The closest functionality available is the C# method `GetValues` in `system.data.sqlclient.SqlDataReader` type. Thus, the method `GetValues` is marked as relevant, but not an exact match.

We then calculated coverage (Cov) as the ratio of the number of methods in a type that TMAP found *atleast* one **relevant** mapping to the total number of source methods in that type. We also calculated the Δ_{Cov} as an increase in the Cov in comparison to results reported by previous approaches [12, 24] as : $\Delta_{Cov} = TMAP_{cov} - Prev_{cov}$. A high Δ_{Cov} value indicates the effectiveness of TMAP in finding API method mappings. Finally, we measured the common methods between the exact mappings suggested by TMAP for a source method with the mappings suggested by previous approaches as $TMAP_{exact} \bigcap Prev_{exact}$. We then calculated the number of new mappings (New) as the number of exact mappings sans the common mappings as : $New = |TMAP_{exact} - (TMAP_{exact} \bigcap Prev_{exact})|$. The value New quantifies the new mappings discovered by TMAP approach in comparison to previous approaches.

RQ2 seeks to quantify the effectiveness of TMAP in leveraging feedback in terms of confirmed mappings for pruning irrelevant mappings in the future TMAP results. We used the mappings discovered by TMAP in RQ1 as confirmed mappings feedback to answer RQ2. Since TMAP feedback operates on the package level, we grouped the mappings in RQ1 based on the source API packages. We then used mappings of one group as feedback and the rest as test subjects to measure the effectiveness of leveraging feedback. We repeated this for all the mappings groups. In particular, we first used mappings of `java.io` group as feedback and then measured the effect in the TMAP output for classes in `java.util` and `java.sql`. We then used mappings of `java.util` group as feedback and then measured the effect in the TMAP output for classes in `java.io` and `java.sql`. Finally, we used mappings of `java.sql` group as feedback and then measured the effect in the TMAP output for classes in `java.io` and `java.util`. Since, all the classes in JavaME-Android belong to same package, we used all of the identified mappings as input and measure the effect on three new classes from the `javax.microedition.lcdui.game` package in JavaME API, namely: 1) `Layer`, 2) `GameCanvas`, and 3) `Sprite`.

We first measured the number of relevant mappings per class prior to leveraging feedback as Rel. We next measured the number of relevant mappings per class after leveraging feedback as $Rel - f$: We next measured the number of methods suggested as mapping candidates per class as $Cand$. TMAP reports ten candidates per method as candidates. Finally, we measured the number of methods suggested as mapping candidates per class after leveraging feedback $Cand_f$.

Next we calculated $Recall_f$ as the ratio of Rel_f to Rel. This value represents the number of relevant mappings retained after leveraging the user input. Higher value of $Recall_f$ indicates effectiveness of TMAP in retaining relevant results while leveraging user feedback. We also calculated $\Delta_{Prunning}$ as the ratio of the number of candidates filtered from the output. Specifically, $\Delta_{Prunning}$ is calculated as $\frac{Cand - Cand_f}{Cand}$. A higher value of $\Delta_{Prunning}$ indicates the effectiveness of TMAP in pruning irrelevant results while leveraging user feedback.

Copyright © 2016 John Wiley & Sons, Ltd.
Prepared using smrauth.cls

DOI: 10.1002/smr

5.3. Results

We next describe our evaluation results to demonstrate the effectiveness of TMAP in leveraging natural language API descriptions to discover method mappings across APIs.

5.3.1. RQ1: Effectiveness of TMAP Table II presents our evaluation results for answering RQ1. The columns 'API' lists the name of source API under 'Source' and target API under 'Target'. The column 'Type' lists the class or interface in source API under consideration for finding mappings in target API. The column 'No. Methods' lists the number of methods in the class or interface under consideration. The column 'Relevant' lists the number of methods for which at least one relevant mapping is reported. The sub-column 'Prev.' reports relevancy numbers by previous approaches. The previous approach for comparison of Java ME-Android mappings is Rosetta [12]. The previous approach for comparison of Java-C# mappings is StaMiner [24]. The sub-column 'TMAP' reports relevancy numbers by TMAP (at least one relevant method in top-ten results). The column 'Exact' lists the number of methods for which an exact mapping is found. The sub-column 'Prev.' reports exact numbers by previous approaches. Since previous approaches do not make a distinction between exact and relevant, we report the same values for both columns. The sub-columns 'TMAP' reports exact numbers by TMAP (at least one exact method mapping in top-ten results). Column 'Δ_{Cov}' lists the ratio of the increase in the number of methods for which a relevant mapping was found by TMAP to the total number of methods in the type.

Our evaluation results indicate that the TMAP on average finds relevant mappings for 57% (Column 'Δ_{Cov}') more methods. For the Java ME-Android mappings, TMAP performs best for `Alert` and `Font` classes from the `javax.microedition.lcdui` package in Java ME API with a 75% increase in the number of methods for which a relevant mapping was found in the Android API. For the Java-C# mappings, TMAP performs best for `Iterator` interface from the `java.util` package in the Java API finding a relevant method in C# API for all the methods. Previous approach StaMiner reports a manually constructed wrapper type as mappings instead. Furthermore, our results also indicate that TMAP found on average exact mappings for 6.5 $((171 - 73)/15)$ more methods per type with a maximum of 21 additional exact mappings for `java.sql.ResultSet` type as compared to previous approaches. We next describe the cases where TMAP did not find any relevant mapping.

A major cause for inadequate performance of TMAP is lack of one-to-one mapping between methods in the source and target API. Often times functionality of a method in a source API is broken down into multiple functions in the target API or vice versa. Although TMAP reports some of the relevant methods, exact mapping may involve a sequence of method calls in the target API which is a limitation of TMAP. In the future, we plan to investigate techniques to deal with such cases.

Another cause of inadequate performance of TMAP is inconsistent use of terminology across different APIs. For instance, TMAP did not find any additional relevant mapping for methods in the `Command` class in Java ME API. In the Java ME API, 'command' is used to refer the user-interface construct 'button'. In the Android API, 'command' is used in more conventional sense of the term. This inconsistent use of terminology causes TMAP to return irrelevant results. When we manually replaced the term 'command' with 'button' in the generated queries, we observed a relevant method appear in top ten results for every method in the `Command` class in Java ME API. However, we refrained from including such modifications to stay true to the TMAP approach for evaluation. In the future, we plan to investigate techniques to automatically suggest alternate keywords.

We also evaluated the effectiveness of TMAP in discovering mappings by switching the source and target API's in Table II (C#-Java mappings and Android-Java ME mappings). Based on results reported in Table II, we identified classes from the C# and Android API that provided the similar functionality to the subject classes/interfaces used in Table II. Since there is not always a strict one-to-one correspondence between classes/interfaces in the APIs, we selected 11 of the most related classes/interfaces. Rosetta [12] and StaMiner [24] do not report on the results of C#-Java mapping and Android-Java ME mappings so we did not compare our results with these approaches.

Copyright © 2016 John Wiley & Sons, Ltd.
Prepared using smrauth.cls

DOI: 10.1002/smr

Table II. Evaluation Results RQ1

S No.	API Source	API Target	Type	No. Methods	Relevant Prev	Relevant TMAP	Exact Prev	Exact TMAP	Δ_{Cov}	Common	New
1	Java ME	Android	javax.microedition.lcdui.Alert	16	3	15	3	7	0.75	0	7
2	Java ME	Android	javax.microedition.lcdui.Canvas	22	5	18	5	10	0.60	0	10
3	Java ME	Android	javax.microedition.lcdui.Command	6	3	3	3	0	0.00	0	0
4	Java ME	Android	javax.microedition.lcdui.Graphics	39	18	36	18	29	0.47	5	24
5	Java ME	Android	javax.microedition.lcdui.Font	16	3	15	3	8	0.75	0	8
6	Java	C#	java.io.File	54	15	37	15	26	0.41	7	19
7	Java	C#	java.io.Reader	10	1	8	1	6	0.70	1	5
8	Java	C#	java.io.Writer	10	2	10	2	10	0.80	1	9
9	Java	C#	java.util.Calendar*	47	0	11	0	5	0.24	0	5
10	Java	C#	java.util.Iterator*	3	0	3	0	1	1.00	0	1
11	Java	C#	java.util.HashMap	17	5	9	5	5	0.24	1	4
12	Java	C#	java.util.ArrayList	28	6	22	6	15	0.58	4	11
13	Java	C#	java.sql.Connection	52	1	28	1	13	0.52	1	12
14	Java	C#	java.sql.ResultSet	187	10	146	10	31	0.73	1	30
15	Java	C#	java.sql.Statement	42	1	21	1	5	0.48	1	4
Total				549	73	382	73	171	0.57**	22	149

*=Previous approach reported a manually constructed class as mapping; **=Average

Prev = previous approach; The previous approach for Java ME-Android mappings is Rosetta [12]; The previous approach for Java-C# mappings is StaMiner. [24]

Copyright © 2016 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

Table III. Evaluation results for reverse-mapping for RQ1

S. No.	API Source	API Target	Type	No. of Mtds	No. of Relevant	No. of Exact
1	C#	Java	system.io.File	7	7 (100)	5 (71.4)
2	C#	Java	system.io.StreamWriter	3	3 (100)	1 (33.3)
3	C#	Java	system.io.StreamReader	6	4 (66.7)	1 (16.7)
4	C#	Java	system.collections.IEnumerator	2	2 (100)	2 (100)
5	C#	Java	system.globalization.Calendar	28	24 (85.7)	18 (64.3)
6	C#	Java	system.data.sqlclient.SqlConnection	7	6 (85.7)	5 (71.4)
7	C#	Java	system.collections.ArrayList	26	22 (84.6)	18 (69.2)
8	C#	Java	system.collections.Hashtable	12	10 (83.3)	7 (58.3)
9	Android	Java ME	android.app.AlertDialog	25	11 (44.0)	1 (04.0)
10	Android	Java ME	android.graphics.Rect	37	13 (35.1)	2 (05.4)
11	Android	Java ME	android.graphics.Canvas	86	46 (53.5)	9 (10.5)
	Total			239	148(61.9)	69(28.9)

Table III presents our evaluation results for discovering aforementioned mappings. The column 'API' lists the name of source API under 'Source' and target API under 'Target'. The column 'Type' lists the class or interface in source API under consideration for finding mappings in the target API. The column 'No. of Mtds' lists the number of methods in the class or interface under consideration. The column 'No. of Relevant' lists the number of methods for which at least one relevant mapping is reported. The column 'No. of Exact' lists the number of methods for which an exact mapping is found.

Our evaluation results indicate that TMAP on average finds relevant mappings for 62% (Column 'No. of Relevant') and 29% (Column 'No. of Exact') exact mappings for the methods in the source API. Grouping by source and target APIs, TMAP finds on average 85.7% relevant and 62.6% exact C#-Java mappings. These results are consistent with the results reported for Java-C# mappings in Table II. In contrast, TMAP finds on average 47.3% relevant and 8.1% exact Android-Java ME mappings.

We further investigated the low effectiveness of TMAP in discovering Android-Java ME mappings. We suspect that since Android API is significantly more advanced and detailed than the older and smaller Java ME API, mappings may not even exist for a lot of methods in Android API. For instance, Android API provides multiple classes/interfaces (and corresponding methods) to handle events and render graphics on a mobile platform. Each type is responsible for handling a specific aspect of functionality in detail. In contrast, Java ME has a broader, more extensible framework. As an example, Android API provides `android.graphics.Rect` class with multiple methods to render rectangles on the screen. In contrast, Java ME API provides a two methods encapsulated within `javax.microedition.lcdui.Graphics` to achieve the same functionality. Furthermore, in the cases when the two APIs have similar types, the methods supported by Android API are more targeted and detailed in terms of functionality supported. Such difference in detail of the supported functionality causes fewer matches for the methods of Android API in Java ME API. In contrast, we find multiple matches for the methods of Java ME API in Android API, as shown in Table II.

5.3.2. RQ2: Improvement by leveraging user feedback To answer RQ2, we compared the output of TMAP with and without leveraging feedback in terms of confirmed mappings. In our experiments, we neither found an improvement nor any detrimental effect while leveraging the feedback to improve the output of Java-C# mappings. We next report on the improvement observed in Java ME-Android mapping. We used the mappings discovered in RQ1 as user feedback to TMAP. We next measured the effectiveness of leveraging feedback on three new classes

Copyright © 2016 John Wiley & Sons, Ltd.
Prepared using smrauth.cls

DOI: 10.1002/smr

Table IV. Evaluation Results for RQ2

S No.	Class Name	Rel	Rel_f	Cand	Cand_f	Recall_f	$\Delta_{Prunning}$
1	Layer	24	20	90	56	0.83	0.36
2	GameCanvas	18	15	60	26	0.83	0.57
3	Sprite	41	39	210	94	0.95	0.55
Total		83	74	360	176	0.89	0.51

from `javax.microedition.lcdui.game` package in JavaME API, namely: 1) `Layer`, 2) `GameCanvas`, and 3) `Sprite`.

Table IV presents our results for Java ME-Android mapping leveraging feedback [‡‡]. Column 'Recall$_f$' lists the effectiveness in retaining the relevant mappings after leveraging the feedback. Column '$\Delta_{Prunning}$' lists effectiveness of TMAP in pruning irrelevant results after leveraging the feedback. Our results indicate that for Java ME-Android mappings, TMAP effectively leverages feedback by pruning an average of 51% irrelevant mapping candidates, while retaining 89% of relevant mappings.

We attribute the lack of any improvement while leveraging feedback for Java-C# to the relatively comparable size of the APIs in terms of number of classes. We also suspect that both Java and C# APIs being programming language APIs and have rich documentation, TMAP is performing well even without leveraging the feedback. In contrast, Java ME that has less than 100 classes, is much smaller and also is a subset of the Android API that has more than 4,000 classes. We think that leveraging feedback will further improve the output of TMAP in cases where the source API and target APIs are significantly different in terms of size. In summary, the results of leveraging user feedback to further improve TMAP output are encouraging; warranting further investigation to improve the feedback strategy by TMAP. In the future, we plan to further investigate efficient feedback strategies, for instance leveraging feedback on the class level instead of package level.

5.3.3. RQ3: Quality of discovered mappings To answer RQ3, we compared the exact mappings discovered by TMAP with the mappings discovered by previous approaches. In Table II, the previous approach for comparison of the Java ME-Android mappings is Rosetta [12], and the previous approach for comparison of Java-C# mappings is StaMiner [24]. Our results show that out of 171 discovered exact mappings, only 22 are in common with previous approaches. We next discuss some of the implications of the results.

Before carrying out this evaluation, we expected that the mappings discovered by TMAP would significantly overlap with the mappings discovered by Rosetta and StaMiner, as these approaches infer mappings from actual source code. Thus, these mappings can be considered as the representative of how developers are actually migrating software. However, the results suggest a low overlap. We manually investigated the possible TMAP specific implications of the observed mismatch.

The results (matches found) comprise of methods from different classes in the target API, reflecting that often there are multiple ways to solve a problem, or to implement a feature using an API. Further, the choice of using multiple APIs gives a developer the flexibility to use different approaches when porting an application from one platform to another.

When more than one match is found for a given source method, the results in TMAP are ranked according to the similarity score, with the more relevant (or exact matches) ranked higher. The ranked set of results helps the developer use the best suited or most appropriate target method in their implementation. This approach is different from earlier approaches [12, 24] that focused on only exact matches between different classes using the number of similar methods as a basis.

[‡‡]Since we neither observed an improvement nor any detrimental effects on Java-C# mapping, we omit Java-C# mapping results for brevity.

Copyright © 2016 John Wiley & Sons, Ltd.
Prepared using smrauth.cls

DOI: 10.1002/smr

Table V. Comparison with Sniff

Query	Method Rank	
	Sniff	TMAP
get active editor window from eclipse workbench	1	1
parse a java source and create ast	1	2
connect to a database using jdbc	1	6
display directory dialog from viewer in eclipse	1	1
read a line of text from a file	1	-
return an audio clip from url	1	1
execute SQL query	2	3
current selection from eclipse workbench	1	1

'-'=No Match in top-10 results.

We also contacted the authors of the Rosetta approach [12] to report the difference in the mappings. Specifically, we inquired why the Rosetta approach does not report any method from the `AlertDialog` class in the Android API as a possible mapping for the `Alert.setString` method in the Java ME API. Rosetta reports the method sequence `Paint.setAlpha CompundButton. setChecked` as one of the likely mappings. In contrast, TMAP discovers the `AlertDialog.setTitle` method in the Android API as a likely mapping.

The lead author of the Rosetta approach responded that they restricted the output of Rosetta to sequences of length up to 2 when inferring mappings (i.e., A→p;q, or A→p). Furthermore, the Rosetta authors count a reported method sequence as a valid mapping if the reported sequence implements some of the functionality of a source method on the target platform. With regards to our query, the Rosetta authors observed that in many of the traces, setting a `string` first involves setting the attributes of the `Paint` (which is used to draw the text), followed by a call to `setText` method, which led them to believe that the sequence `Paint.setAlpha CompundButton. setChecked` could be a likely mapping, if at least in part. Although the author did confirm that technically the `AlertDialog.setTitle` method in Android API as a relevant mapping.

The exchange with the Rosetta approach's lead author points out that the mappings discovered by TMAP generally point to the closest aggregate API, in contrast to the individual smaller API calls that achieve the same functionality. Furthermore, the exchange also demonstrates the reliance of the program analysis approaches on the quality of the code under analysis. In contrast, TMAP relies on the quality of the API method descriptions.

5.3.4. RQ4: Effectiveness in free form queries RQ4 demonstrates the effectiveness of text-mining in general for API related information retrieval tasks. In particular, we show the effectiveness of creating vector space representation of API method descriptions by using free form queries. For evaluating RQ4, we used the same queries as used by Chatterjee et al.'s [7] in evaluation of their approach Sniff. Sniff is targeted towards searching for relevant method snippets from source-code repositories. Sniff first annotates the source code with API document descriptions and uses the hybrid representation of source code to get relevant code fragments. Since TMAP does not report method sequences, we considered a method reported by TMAP as relevant iff:

1. The method is in top-10 results.
2. The method is one of the methods in the code fragment reported by Sniff.

Table V shows the effectiveness of TMAP using free form queries. The column 'Query' lists the query used in the evaluation. The column 'Sniff' lists the ranking of relevant code fragment by the Sniff authors as reported in their work. The column 'TMAP' lists the ranking of first relevant method returned by TMAP.

Copyright © 2016 John Wiley & Sons, Ltd.
Prepared using smrauth.cls

DOI: 10.1002/smr

Our evaluation results show that TMAP returns the relevant method in top-10 results (except one), demonstrating effectiveness of TMAP with free form queries. For the query 'read a line of text from a file' all the reported result methods did support the functionality of reading lines from the file. However, the results reported by TMAP were not reported as relevant code fragment by the Sniff approach.

6. LIMITATIONS AND FUTURE WORK

We next describe the limitations and future work of TMAP, followed by discussions on threats to validity.

A key limitation of the presented work is its reliance on the human developer to confirm or refute mappings. In the future, we plan to extend the TMAP infrastructure to achieve end-to-end automation. Particularly, we plan on using the program-analysis techniques, such as type-analysis proposed in existing approaches [24, 41].

Sometimes the functionality achieved by a method call in a source API is achieved by a sequence of method calls in the destination API, and vice versa. Although TMAP may return individual methods as relevant, TMAP does not provide explicit sequences of method calls as relevant suggestions. In the future, we plan to extend the current text-mining infrastructure to provide method sequences as relevant suggestions when applicable. In particular, we plan to leverage the NLP techniques, such as specification inference [27, 28], to identify method sequences.

From an implementation perspective, TMAP does not take into account the API fields, which limits TMAP's ability in reporting mappings involving API fields. For instance, the functionality achieved by a method call in a source API may be achieved by accessing an API field in the destination API. However, disregarding API fields is a limitation of the current implementation. In future iterations of TMAP implementations,s we plan to include API fields in the indexes as well.

Furthermore, the current implementation of the feedback mechanism uses topic modeling as a mechanism to cluster the API elements together on a package level to prune the irrelevant mappings. Topic modeling has been demonstrated to be useful in many information-retrieval-related software engineering tasks [3, 29]. Although this design choice yielded benefits in Java ME-Android mappings the benefits were not observed in Java-C# mappings. In the future, we plan to further investigate an optimum design of feedback mechanism for effectively pruning irrelevant mappings from the search results. For instance, TMAP currently only leverages the confirmed 'relevant' API mappings to perform pruning. In the future, we plan to leverage the confirmed 'irrelevant' API mappings to further improve our results.

TMAP incorporates a simple camelcase splitting [19] as a means to transform identifiers into meaningful phrases to improve search results. While simple to implement, sometimes naive splitting is not ideal to deal with identifiers containing abbreviations or acronyms. In the future, we will explore alternatives handle such cases. In particular, we are interested in incorporating the concepts from SWordNet [38] to achieve better resolution of camelcase identifiers.

Finally, TMAP operates under the assumption of the availability of the API documents. Thus, TMAP is not applicable in situations where API documents are of low quality or are unavailable altogether. In the future, we plan to extend TMAP infrastructure to work around such situations by integrating with existing source-code-mining-based approaches. Specifically we plan to leverage techniques like code summarization [34] and IR-based approaches like Exoa [18].

Threats to Validity: The primary threat to external validity is the representativeness of our experimental subjects to real-world software. To address this threat, we chose real-world API pairs: 1) Java ME and Android APIs are two Java-based platforms to develop mobile applications; 2) Java and C# APIs are the top object-oriented programming language APIs used for generic application development. The threat can be further minimized by evaluating TMAP on more APIs from different domains. Hopefully, other researchers will emulate our methods to repeat, refute, or improve our results.

Copyright © 2016 John Wiley & Sons, Ltd.
Prepared using smrauth.cls

DOI: 10.1002/smr

Our chief threat to internal validity is the accuracy of TMAP in discovering API mappings. To minimize this threat we compared the mappings inferred by TMAP with the mappings provided by previous approaches. We thank Gokhale et al. [12] for sharing with us the Java ME and Android API mappings inferred by their Rosetta approach. We also thank Nguyen et al. [24] for making their mappings publicly available. Furthermore, the authors did manually identify some of the mappings that could not be compared to previous work. Thus, human errors may affect our results. To minimize the effect, each annotation was independently agreed upon by two authors.

7. RELATED WORK

Language migration is an active area of research [12, 14, 23, 24, 36, 37, 40], with a myriad techniques that have been proposed over time to achieve automation. However, most of these approaches focus on syntactical and structural differences across languages. For instance, Deursen et al. [36] proposed an approach to automatically infer objects in legacy code to effectively deal with differences between object-oriented and procedural languages. However, El-Ramly et al. [10] suggests that most of these approaches support only a subset of APIs for migration. A recently published survey by Robillard et al. [32] provides a detailed overview of techniques dealing with mining API mappings.

Among other works described in [32], Mining API Mapping across different languages [40] (MAM) is most directly related to our work. MAM takes as input a software (S) written in source language and a manually ported version of S (S') written in target language. MAM then applies a technique called "*method alignment*" that pairs the methods with similar functionality across S and S'. These methods are then statically analyzed to detect mappings between source and target language API. Recently, Nguyen et al. [24, 25] proposed StaMiner, an approach that applies statistical-machine-translation-based techniques to achieve language migration. They consider source code as a sequence of lexical tokens (called lexemes) and apply a phrase-based statistical-machine-translation model on the lexemes to achieve migration. While MAM and StaMiner require as input software that has been manually ported from a source to target API (both S and S'), our approach is independent of such requirement. In contrast, TMAP relies on text mining of source and target API method descriptions (that are typically publicly available) to discover likely mappings.

Gokhle et al.'s [12] approach Rosetta addresses the limitations of MAM to infer method mappings. In particular, Rosetta relaxes the constraint of having software that has been manually ported from source to target APIs. Instead, they use functionally similar software in source and target APIs. For instance, they use two different 'tic-tac-toe' game applications in Java ME and Android API not necessarily manually ported. Rosetta approach then requires users to manually execute these applications under identical (or near identical) inputs and collect execution traces. Finally, Rosetta analyses the collected execution traces to infer method mappings. In contrast, TMAP is independent of both the requirements: 1) to have functionally similar applications, 2) to manually execute such applications using similar inputs.

Furthermore, from an infrastructure perspective, TMAP is independent of the programming language or API under consideration. In contrast, program-analysis-based approaches like [12, 24, 40] may need significant efforts to add support for additional APIs and programming languages.

Zheng et al. [39] mine search results of a web search engine, such as Google, to recommend related APIs of different libraries. In particular, they propose heuristics to formulate keywords using the name of the source API, and the name of the target API to query a web search engine. For instance, to search for an equivalent class in C# for the `HashMap` class in Java, a user may manually enter "HashMap C#" in a web search engine. The results are computed one by one and candidates are ranked by relevance, mainly according to the frequency of the appearance of keywords in the query. However, the authors provide only preliminary results, and queries proposed are of a coarse grain. Furthermore, the results are susceptible to influence by the webpages returned by a web search engine. In contrast, TMAP is independent of the web search engine results.

Information retrieval techniques [7, 13, 18, 31] are also being increasingly used in code search. We next describe some relevant approaches. Chatterjee et al.'s [7] approach Sniff annotates the source code with API document descriptions. Sniff then performs additional type analysis on the source

Copyright © 2016 John Wiley & Sons, Ltd.
Prepared using smrauth.cls

DOI: 10.1002/smr

code to rank relevant code snippets. Grechanik et al.'s [13] approach Exemplar uses the text in API documents to construct a set of keywords associated with an API call. Exemplar then uses the keywords list to facilitate query expansion to achieve code search. However, these approaches are targeted towards code search in one API. In contrast, TMAP discovers method mappings across multiple APIs.

Text analysis [8, 19, 27, 28, 41, 42] of API documentation is increasing being used to infer interesting properties from the software engineering perspective. For instance, Zhong et al. [41] employ natural language processing (NLP) and machine learning (ML) techniques to infer resource specifications (rules governing the usage of resources) from API documents. Treude et al. [35] also leverage rule-based NLP approaches to infer action oriented ((programming actions described in documentation) properties from an API document. In contrast, TMAP uses text mining, a comparatively lightweight-approach, instead of sophisticated NLP techniques used in these approaches. Furthermore, TMAP discovers API mapping relations across different APIs for language migration, whereas the previous approaches mine properties of just one API.

8. CONCLUSION

API mapping across different platforms and languages mappings facilitate machine-based migration of an application from one API to another. Thus, tool-assisted discovery of such mappings is highly desirable. In this paper, we presented TMAP: a lightweight text-mining-based approach to infer API mappings. TMAP complements existing mapping inference techniques by leveraging natural language descriptions in API documents instead of relying on source code. We used the TMAP approach to discover API mappings for 15 types across: 1) `Java` and `C#` API, 2) `Java ME` and `Android` API. We demonstrated the effectiveness of TMAP by comparing the discovered mappings with state-of-the-art code-analysis-based approaches. Our results indicate that TMAP on average found relevant mappings for 56% and 57% more methods compared to the Rosetta and the StaMiner approaches, respectively. Our results also show promise in leveraging user feedback to prune irrelevant suggestions in the TMAP output. Furthermore, our results also indicate that TMAP, on average, found exact mappings for 6.5 more methods per class with a maximum of 21 additional exact mappings for a single class as compared to previous approaches.

ACKNOWLEDGEMENT

This work is funded by the USA National Security Agency (NSA) Science of Security Lablet. Any opinions expressed in this report are those of the authors and do not necessarily reflect the views of the NSA. We also thank the Realsearch research group for providing helpful feedback on this work.

REFERENCES

1. Android API Documentation. http://developer.android.com/reference/packages.html.
2. R. Arun, V. Suresh, C. V. Madhavan, and M. N. Murthy. On finding the natural number of topics with latent dirichlet allocation: Some observations. In *Advances in Knowledge Discovery and Data Mining*, pages 391–402. Springer, 2010.
3. D. Binkley, D. Heinz, D. Lawrie, and J. Overfelt. Understanding LDA in source code analysis. In *Proc. of 22nd ICPC*, pages 26–36, 2014.
4. D. M. Blei, A. Y. Ng, and M. I. Jordan. Latent dirichlet allocation. *the Journal of machine Learning research*, 3:993–1022, 2003.
5. J. Bloch. How to design a good API and why it matters. In *Companion to the 21st ACM SIGPLAN OOPSLA*, OOPSLA '06, pages 506–507, 2006.
6. P. F. Brown, V. J. D. Pietra, S. A. D. Pietra, and R. L. Mercer. The mathematics of statistical machine translation: Parameter estimation. *Computational linguistics*, 19(2):263–311, 1993.
7. S. Chatterjee, S. Juvekar, and K. Sen. Sniff: A search engine for Java using free-form queries. In *Proc. of 12th FASE*, pages 385–400. 2009.
8. U. Dekel and J. D. Herbsleb. Improving API Documentation Usability with Knowledge Pushing. In *Proc. 31st ICSE*, pages 320–330, 2009.
9. .NET API Documentation. https://msdn.microsoft.com/en-us/library/gg145045(v=vs.110).aspx.

Copyright © 2016 John Wiley & Sons, Ltd.
Prepared using smrauth.cls

DOI: 10.1002/smr

10. M. El-Ramly, R. Eltayeb, and H. Alla. An experiment in automatic conversion of legacy Java programs to C#. In *Proc. IEEE CSA*, pages 1037–1045, 2006.
11. W. Frakes. Introduction to information storage and retrieval systems. *Space*, 14:10, 1992.
12. A. Gokhale, V. Ganapathy, and Y. Padmanaban. Inferring likely mappings between APIs. In *Proc. 35nd ICSE*, 2013.
13. M. Grechanik, C. Fu, Q. Xie, C. McMillan, D. Poshyvanyk, and C. Cumby. A search engine for finding highly relevant applications. In *Proc. 32nd ICSE*, volume 1, pages 475–484, 2010.
14. A. E. Hassan and R. C. Holt. A lightweight approach for migrating web frameworks. *Inf. Softw. Technol.*, 47(8):521–532, Jun 2005.
15. Java ME API Documentation. http://docs.oracle.com/javame/config/cldc/ref-impl/midp2.0/jsr118/index.html.
16. Java 2 CSharp Translator for Eclipse. http://sourceforge.net/projects/j2cstranslator/.
17. Java API Documentation. http://docs.oracle.com/javase/8/docs/api/.
18. J. Kim, S. Lee, S.-w. Hwang, and S. Kim. Towards an intelligent code search engine. In *Proc. AAAI*, 2010.
19. G. Little and R. C. Miller. Keyword programming in Java. In *Proc. 22nd ASE*, pages 84–93, 2007.
20. Apache Lucene Core. http://lucene.apache.org/core/.
21. C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to information retrieval*, volume 1. Cambridge University Press, 2008.
22. A. K. McCallum. Mallet: A machine learning for language toolkit. http://mallet.cs.umass.edu, 2002.
23. M. Mossienko. Automated Cobol to Java recycling. In *Proc. 7th CSMR*, pages 40–, 2003.
24. A. T. Nguyen, H. A. Nguyen, T. T. Nguyen, and T. N. Nguyen. Statistical learning approach for mining API usage mappings for code migration. In *Proc. 29th ASE*, pages 457–468, 2014.
25. A. T. Nguyen, T. T. Nguyen, and T. N. Nguyen. Migrating code with statistical machine translation. In *Companion Proceedings of the 36th International Conference on Software Engineering*, pages 544–547. ACM, 2014.
26. R. Pandita, R. P. Jetley, S. D. Sudarsan, and L. Williams. Discovering likely mappings between APIs using text mining. In *Proc. 15th SCAM*, pages 231–240, 2015.
27. R. Pandita, K. Taneja, T. Tung, and L. Williams. ICON: Inferring temporal constraints from natural language API descriptions. In *Proc. 32nd IEEE ICSME*, 2016.
28. R. Pandita, X. Xiao, H. Zhong, T. Xie, S. Oney, and A. Paradkar. Inferring method specifications from natural language API descriptions. In *Proc. 34th ICSE*, pages 815–825, 2012.
29. A. Panichella, B. Dit, R. Oliveto, M. Di Penta, D. Poshyvanyk, and A. De Lucia. How to effectively use topic models for software engineering tasks? an approach based on genetic algorithms. In *Proc. of 35th ICSE*, pages 522–531, 2013.
30. API Similarity Project Website. https://sites.google.com/a/ncsu.edu/apisim/.
31. S. P. Reiss. Semantics-based code search. In *Proc. 31st ICSE*, pages 243–253, 2009.
32. M. P. Robillard, E. Bodden, D. Kawrykow, M. Mezini, and T. Ratchford. Automated API property inference techniques. *IEEE Trans. on Software Engineering*, 39(5):613–637, 2013.
33. A. Singhal. Modern information retrieval: A brief overview. *IEEE Data Eng. Bull.*, 24(4):35–43, 2001.
34. G. Sridhara, L. Pollock, and K. Vijay-Shanker. Generating parameter comments and integrating with method summaries. In *Proc. of 19th ICPC*, pages 71–80, 2011.
35. C. Treude, M. P. Robillard, and B. Dagenais. Extracting development tasks to navigate software documentation. *IEEE Trans. on Software Engineering*, 41(6):565–581, 2015.
36. A. Van Deursen and T. Kuipers. Identifying objects using cluster and concept analysis. In *Proc. 21st ICSE*, pages 246–255, 1999.
37. R. C. Waters. Program translation via abstraction and reimplementation. *IEEE Trans. on Software Engineering*, 14(8):1207–1228, 1988.
38. J. Yang and L. Tan. SWordNet: Inferring semantically related words from software context. *Empirical Software Engineering*, 19(6):1856–1886, Dec 2014.
39. W. Zheng, Q. Zhang, and M. Lyu. Cross-library API recommendation using web search engines. In *Proc. 13th ESEC/FSE*, pages 480–483, 2011.
40. H. Zhong, S. Thummalapenta, T. Xie, L. Zhang, and Q. Wang. Mining API mapping for language migration. In *Proc. 32nd ICSE*, pages 195–204, 2010.
41. H. Zhong, L. Zhang, T. Xie, and H. Mei. Inferring resource specifications from natural language API documentation. In *Proc. 24th ASE*, pages 307–318, 2009.
42. H. Zhou, F. Chen, and H. Yang. Developing Application Specific Ontology for Program Comprehension by Combining Domain Ontology with Code Ontology. In *Proc. 8th QSIC*, pages 225 –234, 2008.

Copyright © 2016 John Wiley & Sons, Ltd.
Prepared using smrauth.cls

DOI: 10.1002/smr