

# Inferring Likely Mappings Between APIs using Text Mining

Rahul Pandita<sup>†</sup>, Raoul Jetley<sup>‡</sup>, Sithu D Surarsan<sup>‡</sup>, and Laurie Williams<sup>†</sup>

<sup>†</sup>North Carolina State University, Raleigh, NC, USA

<sup>‡</sup>ABB Corporate Research, Bangalore, KA, India

rpandit@ncsu.edu, abc@xyz.com, williams@csc.ncsu.edu, sudarsan.sd@in.abb.com

## ABSTRACT

Mapping across different platforms/languages Application Programming Interface (API) is highly desirable. Among other reasons, mappings facilitate machine-based migration of an application from one API to another. Such translation is particularly relevant in current context as developers are increasingly migrating applications to diverse platforms/languages to attract and retain users across such platforms/languages. Given a typical platform/language expose a large number of API's for developers to reuse, manually writing these mappings is prohibitively resource-intensive and may be error prone. Recently, researchers proposed to automate the process by mining API mappings from existing codebases. However, these approaches rely on existence of manually ported (or at least functionally similar) code across source and target API's. To address the shortcoming, in this paper we propose SMAM: *text mining* based approach to infer API mappings from the textual description present in API documents. To evaluate our approach, we apply SMAM to infer API mappings across: 1) Java and C# API, 2) J2ME and Android API. We next compare the inferred mappings with state-of-the-art code mining based approaches. Our results indicate that SMAM is effective in inferring API mappings with the more than XX accuracy.

## Categories and Subject Descriptors

H.4 [Information Systems Applications]: Miscellaneous; D.2.8 [Software Engineering]: Metrics—complexity measures, performance measures

## General Terms

Theory

## Keywords

ACM proceedings, L<sup>A</sup>T<sub>E</sub>X, text tagging

## 1. INTRODUCTION

Software is ubiquitous and lately people are increasingly interacting with applications that run on variety of software platforms through their consumer devices on daily basis. For instance, a typical user

may view emails on his/her android or iOS or windowsPhone handheld device such as a mobile phone or a tablet while on the go. The user then can switch to a desktop environment such as Windows or Linux or OSX in an office like setting through either a web browser or a standalone application such as Microsoft Outlook or Mozilla Thunderbird.

To retain users across different platforms, developers are increasingly releasing different versions of their applications. For example, a mobile software developers often release their applications for all the popular mobile platforms such as Android, iOS, and Windows. In context of desktop software, many well-known projects such as JUnit, Hibernate provide multiple versions in different languages, in an attempt to attract developer community to use these libraries across different languages.

However, manually migrating a software from one platform (language) to another can be time consuming and may be error prone. Existing language migration tools such as Java2CSTranslator [10] partially alleviate the problem, as they require a programmer to manually describe how Application Programming Interfaces (APIs) of a source platform(language) maps to API's of the target platform(language). Given a typical platform(language) expose a large number of API's for developers to reuse, manually writing these mappings is prohibitively resource intensive and may result in manual error.

*The goal of this research is to support software developers in migrating an application from one API to another by automatically inferring method mappings across API using text mining on the natural language API method descriptions*

In the literature, there exist approaches that address the problem of finding mapping between software API's using static [29] and dynamic [6] analysis. Recently Nguyen et al. [17] also proposed to apply statistical language translation techniques to achieve language mapping by mining large open source software repositories. However, aforementioned approaches rely on the presence of manually ported (or at least functionally similar) software across source and target API's. Furthermore, accuracy of static analysis and mining based approaches [17, 29] is dependent upon the quality of the code under analysis. Likewise, accuracy of dynamic approaches like [6] is function of the quality and completeness of test case to execute the comprehensive API behavior dynamically.

To address the shortcomings of existing program-analysis based approaches, we propose to use the natural language API method descriptions to infer method mappings across APIs. Our key intu-

ition is: “since the documentation is 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 provides information about how to use a particular method within a class by means of method descriptions. Method descriptions typically describe specifications in terms of the expectations of the method arguments and functionality of method in general.

This paper presents SMAM: An approach that leverages the natural language method descriptions to find the likely mapping. In particular, SMAM proposes to create a vector space model [14,23] of the target API method descriptions. SMAM then queries the vector space model of target API using automatically generated queries from the source API method descriptions. SMAM automates the query generation in source API using the concepts from text mining such as emphasizing (or de-emphasizing) certain keywords over others and querying multiple facets (such as class description, package names etc...).

We pose the following research question : *How accurately can the similarity in the language of API method descriptions be leveraged to infer API Mappings?* To answer our question, we apply SMAM to infer API mappings across: 1) Java and C# API; 2) J2ME and Android API. We next compare the inferred mappings with state-of-the-art code mining based approaches [6, 17]. Our results indicate that SMAM is effective in inferring API mappings with the more than XX accuracy.

In summary, SMAM leverages natural language description of API’s to infer likely mapping thus facilitating cross API migration of applications. As our approach analyzes API documents in natural language, it can be reused independent of the programming language of the library. Our paper makes the following major contributions:

- A technique that effectively infers mapping across source and target API.
- A prototype implementation of our approach based on extending the Apache Lucene [13] and Mallet [15]. An open source implementation of our prototype can be found at our website<sup>1</sup>.
- An evaluation of our approach on J2ME to Android API and Java to C# API

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

## 2. EXAMPLES

We next present an example to motivate our work and list the considerations for applying text mining techniques on API documents. This example is from real Java Platform, Micro Edition (JME) formerly known as J2ME and Android API. Both J2ME and Android use Java as the language of implementation and targeted towards hand-held devices. However, our approach is independent of

<sup>1</sup><http://research.csc.ncsu.edu/ase/projects/pint/>

```
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 method of Graphics class in J2ME

```
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 method of Canvas class in Android

language of implementation and thus can be used for any similar source and target API.

Figure 1 shows the API method description of drawString method from J2ME API. Figure 2 shows the API method description of the Android API equivalent method drawText. Notice the overlap in the language of these two methods. From a human perspective its easier to conclude that the two methods offer similar functionality after reading the descriptions. However, there are lot of intermediate steps to reach that conclusion. For instance, Android API has more than 23,000 public methods. Manually going through each method description is impractical. Provided we are given a keyword based search framework (a very basic text mining infrastructure) for querying the Android API method descriptions, the next step is: *what does the query look like?*

Table 1 lists some of the queries we tried on an index of Android API method descriptions to find a mapping for the J2ME method drawString shown in Figure 1. Column “Query” describes the query we executed on the index. Column “Hits” lists the number of matches found, and Column “Relevant 10” lists the rank of the first relevant method in top ten results. For instance, when we searched for the class name “Graphics” in Android API we did not get any hits. We also did not get any hits for when we searched for method name “drawString”.

When we searched for the method signatures we got a 23,547 hits

**Table 1: Query Results**

S No.	Query	Hits	Relevant 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

(almost all methods in Android API), mostly because of the confounding effects of keywords such as “public”. Most of the methods signatures will have the keyword “public”. Although the ranking mechanism did push methods with most keyword matches first, we did not find a relevant method in top 10. Likewise the method descriptions as a whole or in parts also did not yield better results. In this case a combination of various attributes such as method name split in camel case, keywords from class description and method description resulted in the Android API equivalent method `drawText` shown in Figure 2 in the top ten results.

The examples demonstrates difficulties faced by simple keyword-based searches and text mining in general to find likely method mappings. In general, there are following difficulties in applying text mining approaches to API documents :

1. *Confounding effects.* Certain method names have an confounding effects. For instance `toString()`, `get()`, `set()` are too generic and tend to have similar descriptions across different method definitions. These generic methods often cause interference with the output of text mining based approaches.
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 empathizes on the specific type of file. The problem is to automatically identify the importance of a term.
3. *Structure.* API documents are not flat contiguous text blobs. They often have well defined structure, that is often shared by method descriptions. Ignoring the structure may cause ineffective queries negatively affecting the results. This is addressed by querying individual elements (such as class description, class names, method names, method descriptions) and then aggregate the results. The problem is to effectively aggregate the results.

### 3. RELATED WORK

Language migration has been an active area of research [8, 16, 26, 27], with 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. [26] 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. [4]’s experience points out that most of these approaches support only a subset of API’s for migration. Another recently published survey by Robillard et al. [22] provides

a detailed overview of techniques dealing with mining API mappings. Among other works described in [22], Mining API Mapping (MAM) [29] is most directly related to our work.

MAM mines API mapping relations across different languages for language migration, however there is a significant difference between MAM and SMAM. MAM takes into input a software  $S$  written in source language and manually ported version of  $S'$  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. While MAM relies on existence of software that has been ported manually from a source to target API, our approach is independent of such requirement. In contrast, our approach relies on text mining of source and target API document to achieve the same. Furthermore, we demonstrate that our approach also infers equally good mappings if not better **pending evaluation**.

Gokhle et al.’s [6] 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 a source to target API. Instead, they use functionally similar software in source and target API. For instance, they use two different ‘tic-tac-toe’ game applications in J2ME and Android API not necessarily manually ported. They then manually execute these applications under identical (or near identical) inputs and collect execution traces. Finally they analyze the collected execution traces to infer method mappings. In contrast our approach is independent of both the requirements: 1) to have functionally similar applications, 2) to manually execute such applications using similar inputs.

Recently Nguyen et al. [17, 18] proposed to apply statistical machine translation based techniques to achieve language migration. Their approach builds upon a previous result [9] that demonstrates the effectiveness of using a n-gram model to predict the next token in software source file given a large corpus of software source files to learn from. Similar to MAM [29] approach they need a software  $S$  written in source language and manually ported version of  $S'$  written in target language. They then consider source code as a sequence of lexical tokens and apply a phrase-based statistical machine translation model on the lexemes of those tokens. In contrast our approach is independent of such requirement (presence of  $S$  and  $S'$ ).

Furthermore, from an infrastructure perspective, SMAM is independent of the programming language or API under consideration. In contrast program analysis based approaches like [6, 17, 29] may need significant efforts for adding support to additional API’s.

Zheng et al. [28] mine search results of web search engines 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 target API to query 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 their frequency of the appearance of keywords in the query. This work provides only preliminary results and queries proposed are of a coarse grain. Furthermore, the results are susceptible to influence by the outcome of the web search engines. In contrast, our approach is independent of the web search engine. Furthermore, the queries used

in our approach are more sophisticated than the heuristics proposed by their approach.

Information retrieval techniques [2,7,11,21] are also being increasingly used in Code Search. We next describe some relevant approaches. Sniff [2] annotates the source code with API document descriptions. Sniff then performs additional type analysis on the source code to rank relevant code snippets. Exemplar [7] uses the text in API documents to construct a set of keywords associated with an API call. Exemplar then uses the keywords list to perform query expansion to achieve code search. However, these approaches are targeted towards code search in a single API. In contrast SMAM works on inferring mappings across multiple APIs.

Text analysis [3,12,19,30,31] of API documentation is increasing being used to infer interesting properties. We next describe most relevant approaches.

**API property inference** Zhong et al. [30] employ NLP and ML techniques to infer resource specifications from API documents. They define resource specifications as the rules governing the usage of resources such as File. Recently Treude et al. [25] leverage rule-based NLP approaches to infer action (programming task) oriented properties from an API document. They describe tasks as specific programming actions that have been described in the documentation. First, SMAM uses text mining instead of sophisticated NLP techniques used in these approaches. Second, SMAM infers API mapping relations across different API for language migration, whereas the previous approaches mine API properties of a single API to detect defects or to assist programming. Finally, the performance of the preceding ML-based approaches is dependent on the quality of the training sets used for ML. In contrast, our approach is independent of such training set and thus can be easily extended to target respective problems addressed by these approaches.

## 4. BACKGROUND

Text mining is a broad research area including but not limited to the techniques facilitating retrieval/manipulation of useful information from a large corpus to text. It is claimed that more than 80% of business intelligence is captured in unstructured data **need citation**. A large part of this is in documents containing narrative text, such as reports, presentations, web data, design documents, etc. Text mining is used to unearth the knowledge buried in these documents through a combination of extraction and analytical methods.

As opposed to traditional data mining, text mining analyzes free-form text spread across a large number of documents (rather than data localized and maintained within a database). In order to analyze this data, text mining uses concepts from traditional data analytics, natural language processing, and data modeling.

We next introduce the concepts from text mining that have been used in the presented approach.

**Indexing** [5, 14]: Indexing is the process of extracting text data from source documents and storing them in well-defined indexes. Indexing process constitutes of following steps.

A document is divided into constituent unit, known as tokens, based on a well-defined criterion. A token is usually an individually identifiable unit of the document (such as a word) and relates to the individual terms stored in the index. Once the tokens within each document are identified, they are added to the index, with the cor-

responding link to the document and associated term frequencies. An optional pre-processing step further assists with indexing: such as removing stop-words, grouping similar words together.

Among various indexing strategies, use of inverted file indexing [5] is well suited for large document collections. In its simplest form inverted file index maintains the mapping of terms and their location in a text collection. For instance a text document can be thought of as a collection of  $m$  words. It is made up of a sequence of  $n$  unique words such that  $n \leq m$ . The number  $n$  is usually far less than  $m$  as most of the words are repeated while forming a document. For instance the word “the” is repeated many times in this paper. The set of unique words within an index forms the “Term List”  $v$  of the index. If a pointer (say numeric location) is associated with each word in  $v$  to the location of that word in text document, the resultant data structure is a form of inverted file index. As the document collection grows, the number of documents matching a word in the index becomes sparser.

The index is oftentimes annotated with the information of the frequency of occurrence of each term in the document. This representation of a document as a vector of frequencies of terms is referred to as *vector space model* [5,23].

**TF-IDF Similarity** [14]: Term-frequency inverse-document-frequency ( $\text{tf-idf}$ ) 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  $\text{tf-idf}$  weight increases proportionally to frequency of the occurrence of a term in a document, however the weight is also offset by the frequency of occurrence the term in the corpus.

**Cosine Similarity** [23]: In mathematics Cosine similarity is a numerical statistic to measure the similarity between two vector. Cosine similarity is defined as a dot product of magnitude of two vectors. In context of text mining, the Cosine similarity is used to capture the similarity between two documents represented as term frequency vectors.

**Topic Modelling:** Latent Dirichlet Allocation (LDA) [1, 20] is an Information Retrieval (IR) model to fit a probabilistic model on the term occurrences in a corpus of documents. Given a corpus of  $n$  documents, a dictionary is created. The dictionary consists a list of all the unique terms ( $m$ ) occurring in all  $n$  documents. Next, a term-to-document matrix  $M$  of size ( $m \times n$ ) is generated, where each row represents all the terms occurring the corpus and columns representing the all the documents in the corpus. Each cell of  $M$  ( $M[i, j]$ ) contains the weight of the  $i^{th}$  term in  $m$  in the  $j^{th}$  document in  $n$ . In the simplest implementation  $\text{tf-idf}$  is used to calculate the weight of a term in a document.

Next LDA transforms term-to-document matrix  $M$  into a topic-to-document matrix  $\theta$  of size ( $k \times n$ ), where each row represents the topics occurring in the corpus and columns representing the all the documents in the corpus. This transformation is achieved by identifying latent variables (topics) in the documents. The parameter  $k$  is independently provided as an argument to the LDA algorithm. Each cell of  $\theta$  ( $\theta[i, j]$ ) contains the probability of the  $i^{th}$  topic in  $k$  occurring in the  $j^{th}$  document in  $n$ . Since  $k \ll m$ , LDA is a mapping of the documents from the term space  $m$  to the topic space  $k$  [20].

1.  $k$ , the total number of topics to be extracted from a corpus of

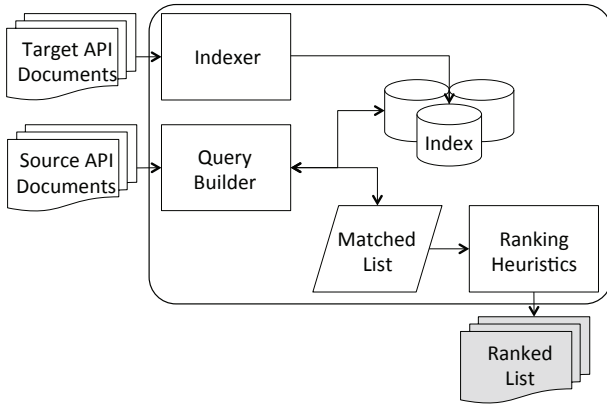


Figure 3: Overview of sMAM approach

Table 2: Extracted Fields

S No.	Name	Description
1	Type Name	Enclosing class/interface name of the method
2	Package Name	Package name of the enclosing class
3	Method Name	Name of the method
4	Method Modifiers	Access modifiers of the method
5	Exceptions Thrown	Name of the exceptions thrown by the method
6	Parameters	Name and type information of parameters
7	Class Description	Description of enclosing type
8	Method Description	Description of the method

documents.

2.  $n$ , the total number of gibbs iteration, where a single iteration involves a gibbs sampler sampling a topic for each term occurring in the corpus.
3.  $\alpha$ , this parameter affects the distribution of topics across documents. A high  $\alpha$  means that each document is likely to contain a mixture of most of the topics. Conversely, a low  $\alpha$  means each document is likely to contain mixture of fewer or one topic.
4.  $\beta$ , this parameter affects the distribution of terms in each topic. A high  $\beta$  means that each topic is likely to contain a uniform distribution of terms. Conversely, a low  $\beta$  means terms are not uniformly distributed across topics.

## 5. APPROACH

We first preprocess the API documents

In particular we extract the fields listed in Table 2 fields from API documents

We perform the following pre-processing

1. **Lowercase** Convert the text to lowercase
2. **Split Camelcase** Split the camelcase notation of the class name and method name

**Class Name:** java util iter  
**Class 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 elementrather than throw except  
 \* Names are Camel-case split

Figure 4: Query

3. **Split package Notation** Split the package notation
4. **Remove href and other javadoc identifiers** remove the @Code, @link... like Javadoc identifiers

First we parse the API documents of source and target API library and store it in intermediate representation for analysis. In particular, we extract class, interface and corresponding method descriptions from API documents. Second, we then create a term vector representation of each class description and method descriptions. Vector space model or term vector model is an algebraic model for representing text documents (and any objects, in general) as vectors of identifiers, such as, for example, index terms. In our case, each word is considered as a term barring the stop words such as (a, the, and ...).

Finally we use the term vector representation of a method description in a source API to query term vector representation of the the method description in target API. The results are ranked based on the Term Frequency-Inverse Document Frequency (TF-IDF) measure. TF-IDF is a numerical statistic which reflects how important a word is to a document in a collection or corpus. It is often used as a weighting factor in information retrieval and text mining.

We next build indexes

Lucene Indexes

using Standard English Analyzer

following fields:

Class Base Name Camel Case Split

Class Description

Method base Name Camel Case Split

Method Description

Query formulation:

heuristics 1 : first two sentences are provide a reasonable approximation of the most important keywords of an method

heuristics 2 : first paragraph provides a reasonable approximation of the most important keywords of a class.

## 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 5: hasNext method of Iterator**

**Table 3: Comparison with SNIFF**

Query	Relevant Mtd rank	
	SNIFF	sMAM
get active editor window from eclipse workbench	1	1
parse a java source and create ast	1	1
connect to a database using jdbc	1	1
display directory dialog from viewer in eclipse	1	1
read a line of text from a file	1	1
return an audio clip from url	1	1
execute SQL query	2	1
current selection from eclipse workbench	1	1

## 6. EVALUATION

We conducted an evaluation to assess the effectiveness of our approach. In our evaluation, we address three main research questions:

- **RQ1:** What are the precision and recall of our approach in identifying API mappings?
- **RQ2:** How do the mappings inferred by our approach compare with the human written mappings?

### 6.1 Subjects

We used the API documents of the following library pairs as subjects for our evaluation.

**J2ME-Android.** Java Platform, Micro Edition, or Java ME, is a Java platform designed for embedded systems (mobile devices are one kind of such systems). Target devices range from industrial controls to mobile phones (especially feature phones) and set-top boxes. Java ME was formerly known as Java 2 Platform, Micro Edition (J2ME).

Android is a Linux-based operating system designed primarily for touchscreen mobile devices such as smartphones and tablet computers.

#### Java -C#

Effectiveness in free form queries:

Discussion with Authors

None of the mapping methods point to AlertDialog Class in Android that seems to be the class implementing alert functionality.

**Table 4: Comparison with Rosetta**

S. No.	Method	sMAM Rank
1	Alert.setTimeout	
2	Alert.setType	
3	Canvas.getHeight	
4	Canvas.getWidth	
5	Canvas.repaint	
6	Canvas.serviceRepaints	
7	Canvas.setFullScreenMode	
8	Command.getCommandType	
9	Command.getLabel	
10	Display.getCurrent	
11	Display.getDisplay	
12	Display.setCurrent	
13	Displayable.addCommand	
14	Displayable.getHeight	
15	Displayable.getWidth	
16	Displayable.removeCommand	
17	Font.charWidth	
18	Font.getHeight	
19	Font.stringWidth	
20	Form.addCommand	
21	Form.setCommandListener	
22	game.GameCanvas.getHeight	
23	game.GameCanvas.getWidth	
24	game.GameCanvas.repaint	
25	game.GameCanvas.serviceRepaints	
26	game.GameCanvas.setFullScreenMode	
27	game.Layer.getHeight	
28	game.Layer.getWidth	
29	game.Layer.setPostion	
30	game.Sprite.paint	
31	game.Sprite setFrame	
32	game.Sprite.setPosition	
33	game.Sprite.setRefPixelPosition	
34	Graphics.clipRect	
35	Graphics.drawArc	
36	Graphics.drawChar	
37	Graphics.drawImage	
38	Graphics.drawLine	
39	Graphics.drawRect	
40	Graphics.drawString	
41	Graphics.fillArc	
42	Graphics.fillRect	
43	Graphics.fillRoundRect	
44	Graphics.fillTriangle	
45	Graphics.getClipHeight	
46	Graphics.getClipWidth	
47	Graphics.getClipX	
48	Graphics.getClipY	
49	Graphics.setClip	
50	Graphics.setColor	
51	Graphics.setFont	
52	Image.createImage	
53	Image.getGraphics	
54	Image.getHeight	
55	Image.getWidth	

Furthermore, consider the first-mapping

```
Alert.setString Paint.setAlpha; CompundButton.setChecked
```

None of the mapped methods set text in the alert. How to interpret the mapping in such cases?

Amruta's Response :

First, a note of clarification. As explained in the paper, we restricted ourselves to sequences of length up to 2 when inferring mappings (i.e., A->p;q, or A->p). So this might mean that p-q only implements some of the functionality of A on the target platform, and we still count this as a valid mapping.

With that said, let me answer your questions: Let's consider the mapping: "Alert.setString->Paint.setAlpha; CompundButton.setChecked". In many of our traces, we observed that setting a string first involves setting the attributes of the paint (with is used to draw the text), followed by a call to setText method, which led us to believe that the sequence "Paint.setAlpha; CompundButton.setChecked" could be a likely mapping, if at least in part. The same reasoning applies to mappings of the other two methods in your list.

Technically, you're right that "Alert" related methods would be implemented using methods of the "AlertDialog" class. However, "Alert" is used to display messages, and we considered Android methods/sequences that could also be used to display messages as valid mappings. These methods/sequences may not strictly be considered "AlertDialogs" in the Android sense, but can still be part of rendering a text message on the screen.

## 7. DISCUSSION AND FUTURE WORK

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

A key limitation of the presented work is reliance on the human developer to confirm or refute mappings. In future, we plan to extend the SMAM infrastructure to achieve an end-to-end automation. Particularly, we plan on using the program analysis techniques such as type-analysis proposed in existing approaches [17, 30].

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 SMAM may return individual methods as relevant, SMAM does not provide explicit sequences of method calls as relevant suggestions. In 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 [19] to identify method sequences.

From an implementation perspective, SMAM does not take into account the API fields. This limits sMAMs 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 a API field in destination API. However, this is a limitation of the current implementation and in future iterations of SMAM we plan to include API fields in the indexes.

Finally, SMAM operates under the assumption of the availability of the API documents. Thus SMAM is not applicable in situations where API documents are low quality or are unavailable altogether.

In future, we plan to extend SMAM infrastructure to workaround such situations by integrating with existing source code mining based approaches. Specifically we plan to leverage techniques like code summarisation [24] and IR based approaches like Exoa [11].

## 7.1 Threats to Validity

The primary threat to external validity is the representativeness of our experimental subjects to the real world software. To address this threat we chose real world API pairs: 1) J2ME 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 SMAM on more APIs from different domains.

Our chief threat to internal validity is the accuracy of SMAM in identifying API mappings. To minimize this threat we compared the mappings inferred by SMAM with the mappings provided by previous approaches. We thank Gokhale et al. [6] for sharing with us the J2ME and Android API mappings inferred by their approach. We also thank Nguyen et al. [17] for making their mappings publicly available. Furthermore, first two did manually identify some of the mappings that could not be compared to previous work. Thus, human errors may affect our results. To minimized the effect, first two authors randomly sampled and inspected each others work.

## 8. CONCLUSION

API mapping across different platforms/languages mappings facilitate machine-based migration of an application from one API to another. Thus automated inference of such mappings is highly desirable. In this paper, we presented sMAM: a lightweight text-mining based approach to infer API mappings. SMAM complements existing mapping inference techniques by leveraging natural language descriptions in API documents instead of relying on existence of manually ported (or at least functionally similar) code across source and target API's. We demonstrated the effectiveness of sMAM by comparing the inferred mappings with state-of-the-art code mining based approaches. Our results indicate that SMAM is effective in inferring API mappings (from over human-annotated API sentences) with the more than XX accuracy.

## 9. ACKNOWLEDGMENTS

This work is supported by xx. Any opinions expressed in this report are those of the author(s) and do not necessarily reflect the views of XX. Finally, we thank the Realsearch research group for providing helpful feedback on this work.

## 10. REFERENCES

- [1] D. M. Blei, A. Y. Ng, and M. I. Jordan. Latent dirichlet allocation. *the Journal of machine Learning research*, 3:993–1022, 2003.
- [2] S. Chatterjee, S. Juvekar, and K. Sen. Sniff: A search engine for java using free-form queries. In *Fundamental Approaches to Software Engineering*, pages 385–400. Springer, 2009.
- [3] U. Dekel and J. D. Herbsleb. Improving API Documentation Usability with Knowledge Pushing. In *Proc. 31st ICSE*, pages 320–330, 2009.
- [4] 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.

Table 5: Comparison with Tien Nyugen

S.No.	Source Class	Target Class	SMAM Ranking
1	java.util.concurrent.atomic.AtomicBoolean		
2	java.util.jar.JarInputStream		
3	java.util.jar.Attributes		
4	java.util.jar.Manifest		
5	java.util.LinkedHashMap	System.Collections.Hashtable	
6	java.util.AbstractCollection	System.Collections.ICollection	
7	java.util.Date	System.DateTime	
8	java.util.HashSet	ILOG.J2CsMapping.Collections.Hashtable	
9	java.util.SortedSet	ILOG.J2CsMapping.Collections.SortedSet	
10	java.util.Set	ILOG.J2CsMapping.Collections.ISet	
11	java.util.TreeSet	ILOG.J2CsMapping.Collections.SortedSet	
12	java.util.BitSet	ILOG.J2CsMapping.Collections.BitSet	
13	java.util.Hashtable	System.Collections.Hashtable	
14	java.util.TreeMap	System.Collections.SortedList	
15	java.util.SortedMap	System.Collections.SortedList	
16	java.util.Iterator	ILOG.J2CsMapping.Collections.IIterator	
17	java.util.IdentityHashMap	System.Collections.Hashtable	
18	java.util.Vector	System.Collections.ArrayList	
19	java.util.WeakHashMap	System.Collections.Hashtable	
20	java.util.StringTokenizer	ILOG.J2CsMapping.Util.StringTokenizer	
21	java.util.EventObject	ILOG.J2CsMapping.Util.EventObject	
22	java.util.Locale	System.Globalization.CultureInfo	
23	java.util.Map.Entry	System.Collections.DictionaryEntry	
24	java.util.UUID	System.Guid	
25	java.util.EventListener	ILOG.J2CsMapping.Util.IEventListener	
26	java.util.AbstractList	System.Collections.ArrayList	
27	java.util.TimerTask	ILOG.J2CsMapping.Util.ITimerTask	
28	java.util.Timer	ILOG.J2CsMapping.Util.ITimer	
29	java.util.EmptyStackException	System.Exception	
30	java.util.Collections	ILOG.J2CsMapping.Collections.Collections	
31	java.util.ListIterator	ILOG.J2CsMapping.Collections.IListIterator	
32	java.util.HashMap	System.Collections.Hashtable	
33	java.util.Calendar	ILOG.J2CsMapping.Util.Calendar	
34	java.util.ResourceBundle	System.Resources.ResourceManager	
35	java.util.Properties	ILOG.J2CsMapping.Util.Properties	
36	java.util.Stack	System.Collections.Stack	
37	java.util.MissingResourceException	System.Resources.MissingManifestResourceException	
38	java.util.Comparator	System.Collections.IComparer	
39	java.util.ArrayList	System.Collections.ArrayList	
40	java.util.Collection	System.Collections.ICollection	
41	java.util.Map	System.Collections.IDictionary	
42	java.util.ConcurrentModificationException	System.InvalidOperationException	
43	java.util.Arrays	System.Array	
44	java.util.Currency		
45	java.util.NoSuchElementException	System.InvalidOperationException	
46	java.util.Enumeration	ILOG.J2CsMapping.Collections.IIterator	
47	java.util.LinkedList	ILOG.J2CsMapping.Collections.LinkedList	
48	java.util.GregorianCalendar	ILOG.J2CsMapping.Util.GregorianCalendar	
49	java.util.Random	System.Random	
50	java.util.PropertyResourceBundle		
51	java.util.List	System.Collections.IList	
52	java.util.regex.Pattern	ILOG.J2CsMapping.Text.Pattern	
53	java.util.regex.Matcher	ILOG.J2CsMapping.Text.Matcher	



- [5] W. Frakes. Introduction to information storage and retrieval systems. *Space*, 14:10, 1992.
- [6] A. Gokhale, V. Ganapathy, and Y. Padmanaban. Inferring likely mappings between APIs. In *Proc. 35nd ICSE*, 2013.
- [7] M. Grechanik, C. Fu, Q. Xie, C. McMillan, D. Poshyanyk, and C. Cumby. A search engine for finding highly relevant applications. In *Software Engineering, 2010 ACM/IEEE 32nd International Conference on*, volume 1, pages 475–484. IEEE, 2010.
- [8] A. E. Hassan and R. C. Holt. A lightweight approach for migrating web frameworks. *Inf. Softw. Technol.*, 47(8):521–532, Jun 2005.
- [9] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu. On the naturalness of software. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 837–847. IEEE, 2012.
- [10] Java 2 CSharp Translator for Eclipse. <http://sourceforge.net/projects/j2cstranslator/>.
- [11] J. Kim, S. Lee, S.-w. Hwang, and S. Kim. Towards an intelligent code search engine. In *AAAI*, 2010.
- [12] G. Little and R. C. Miller. Keyword programming in Java. In *Proc. 22nd ASE*, pages 84–93, 2007.
- [13] Apache Lucene Core. <http://lucene.apache.org/core/>.
- [14] C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to information retrieval*, volume 1. Cambridge university press Cambridge, 2008.
- [15] A. K. McCallum. MALLET: A machine learning for language toolkit. 2002.
- [16] M. Mossienko. Automated Cobol to Java recycling. In *Proc. 7th CSMR*, pages 40–, 2003.
- [17] 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 *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 457–468. ACM, 2014.
- [18] 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.
- [19] 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*, 2012.
- [20] A. Panichella, B. Dit, R. Oliveto, M. Di Penta, D. Poshyanyk, and A. De Lucia. How to effectively use topic models for software engineering tasks? an approach based on genetic algorithms. In *Proc. 35th ICSE*, pages 522–531, 2013.
- [21] S. P. Reiss. Semantics-based code search. In *Proc. 31st ICSE*, pages 243–253, 2009.
- [22] 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.
- [23] A. Singhal. Modern information retrieval: A brief overview. *IEEE Data Eng. Bull.*, 24(4):35–43, 2001.
- [24] 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.
- [25] C. Treude, M. Robillard, and B. Dagenais. Extracting development tasks to navigate software documentation.
- [26] A. van Deursen and T. Kuipers. Identifying objects using cluster and concept analysis. In *Proc. 21st ICSE*, pages 246–255, 1999.
- [27] R. C. Waters. Program translation via abstraction and reimplementation. *IEEE Trans. on Software Engineering*, 14(8):1207–1228, 1988.
- [28] W. Zheng, Q. Zhang, and M. Lyu. Cross-library API recommendation using web search engines. In *Proc. 13th ESEC/FSE*, pages 480–483, 2011.
- [29] 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.
- [30] 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.
- [31] H. Zhou, F. Chen, and H. Yang. Developing Application Specific Ontology for Program Comprehension by Combining Domain Ontology with Code Ontology. In *Proc. 8th QSI*, pages 225 –234, 2008.