Empirical Software Engineering - Authors' self-archived version
The final publication is available at Springer via
http://dx.doi.org/10.1007/s10664-014-9323-y

Recommending Reference API Documentation

Martin P. Robillard · Yam B. Chhetri

Received: date / Accepted: date

Abstract Reference documentation is an important source of information on API usage. However, information useful to programmers can be buried in irrelevant text, or attached to a non-intuitive API element, making it difficult to discover. We propose to detect and recommend fragments of API documentation potentially important to a programmer who has already decided to use a certain API element. We categorize text fragments in API documentation based on whether they contain information that is *indispensable*, *valuable*, or neither. From the fragments that contain knowledge worthy of recommendation, we extract word patterns, and use these patterns to automatically find new fragments that contain similar knowledge in unseen documentation. We implemented our technique in a tool, Krec, that supports both information filtering and discovery. In an evaluation study with randomly-sampled method definitions from ten open source systems, we found that with a training set derived from about 1000 documentation units, we could issue recommendations with about 90% precision and 69% recall. In a study involving ten independent assessors, indispensable knowledge items recommended for API types were judged useful 57% of the time and potentially useful an additional 30% of the time.

Keywords Application Programming Interfaces · API Documentation · Text Classification · Natural Language Processing · Recommendation Systems

1 Introduction

Application Programming Interfaces (APIs) are a means of code reuse. They provide an interface to features and functionality in existing frameworks and libraries. Using APIs, however, often involves significant challenges (Robillard and DeLine, 2011; Stylos and Myers, 2008).

When questions arise about how to use an API correctly or efficiently, programmers naturally expect to find information in the API's reference documentation (Robillard and

M.P. Robillard and Y.B. Chhetri
School of Computer Science
McGill University
Montréal, QC, Canada
E-mail: {martin@cs,yam.chhetri@mail}.mcgill.ca

DeLine, 2011). While good reference documentation will often contain such information, it can become difficult to discover and access given the large size and repetitive nature of the documentation resources. This general challenge can be described in terms of two separate problems: *information filtering* and *information discovery*.

Information filtering relates to the burden of having to sift through large amounts of irrelevant information, e.g., because of legacy information, boilerplate text, or because it is intended for another audience such as novice programmers. Indeed, Maalej and Robillard (2013) found that the documentation units associated with 53.5% of the class members in the .NET 4.0 framework contain obvious information of little value (for the JDK 6 this number is 45.5%). We note that solutions to the information filtering problem do not necessarily involve deleting or hiding information judged irrelevant. Instead, the goal of the information filtering task is to surface the most important information. How this information is presented is an implementation decision.

For API reference documentation, the problem of information discovery relates to the fact that useful information may be attached to an element a reader would not intuitively access. This problem has been discussed extensively in the literature, and has motivated several other research projects (Dekel and Herbsleb, 2009; Duala-Ekoko and Robillard, 2012; Stylos and Myers, 2008). Section 2 illustrates the need to support both information filtering and information discovery.

To address the challenges described above, we wanted to determine to what degree it was possible to identify what fragments of text in an API's reference documentation should be read by a programmer *who has decided to use this element as part of a task, but independently of the actual task.* This constraint provides a specific criterion that we can use to judge the usefulness of individual pieces of information in reference documentation. Although the notion of relevance often includes context-specificity (e.g., details of the task or characteristics of the user), our initial goal was to investigate how to create a context-independent knowledge base from API documentation. In practice, recommendations derived from such a knowledge base can always be filtered based on user profiles.

Our initial exploration of the idea of recommending API documentation led to two main contributions toward a general solution.

First, using a grounded approach, we created a *coding guide* to manually but reliably classify text fragments into three distinct categories: those that are *indispensable*, those that are *valuable*, and those that are *less important* to a programmer who has already identified the corresponding element as relevant to a task. We validated our coding guide by measuring inter-coder agreement on classification performed independently.

Second, we devised a technique to identify *indispensable* and *valuable* pieces of information in documentation units. We call these pieces of information *knowledge items (KIs)*. The technique relies on the semi-automatic identification of word usage patterns in relevant information, assisted by natural language processing techniques, and then uses a pattern-matching approach to automatically find instances of these patterns in unseen documentation.

As part of this work, we have also developed an Eclipse plug-in, *Krec*, that recommends the first two categories of information from the reference documentation of the API elements used in a block of code.

We evaluated the approach end-to-end in the context of software development in Java using the JDK reference documentation as provided by the output of the Javadoc tool. We evaluated the approach with sample Java code blocks extracted from ten open source systems. Our evaluation shows that with a training set consisting of 556 pieces of useful information extracted from over 1000 manually-inspected documentation units, we can issue

recommendations in simulated operating conditions with about 90% precision and 69% recall. Furthermore, the recommended knowledge represents the filtering out of about 86% of the less important reference documentation associated with a programming (recommendation) context. We further verified the recommendations on code blocks from a popular book that recommends efficient ways of programming in Java (Bloch, 2008), and found that for 6 out of 8 possible cases, *Krec* was able to match the recommendation from the documentation with those expected in the book. Finally, we conducted a study where we asked ten independent assessors to rate the usefulness of knowledge items tagged as indispensable by our approach that were recommended for API types. The participants rated the recommended knowledge items as useful in 57% of cases, and as potentially useful in another 30% of cases.

The rest of this article is organized as follows. Section 2 illustrates the need for information filtering and information discovery when using API reference documentation. Section 3 defines our concept of knowledge item and explains our development of a coding guide that is the instrument we use to classify text fragments in reference documentation. Section 4 describes our automated technique for detecting and recommending useful information in reference documentation, and Section 5 reports on its performance. Section 6 discusses the related work and Section 7 presents our conclusions.

2 Motivating Examples

The following situations illustrate how a recommendation-based model for accessing API documentation supports both information filtering and discovery.

Cloning Objects

In Java, *cloning* is an important mechanism that can be used to make a copy of an object without statically specifying its run-time type. Cloning in Java is a complex mechanism that is supported both by language constructs (interface implementation) and by API elements (such as the `Cloneable` interface and the `Object.clone()` method).

The reference documentation for `Object.clone()` provides both a theoretical specification of the behavior of the method, as well as constraints on how it should be overwritten. The description of the method comprises 344 words in 16 sentences.

Once a developer has assimilated the basic theory behind the Java cloning mechanism, the value of this documentation unit lies mostly in the important implementation guidelines it provides, such as the following sentences:

> *By convention, the returned object should be obtained by calling* `super.clone`.
>
> or,
>
> *By convention, the object returned by this method should be independent of this object (which is being cloned).*

However, without explicit support, the only mechanism available to a developer to remember these kinds of directives is to re-read the documentation of `Object.clone()` to find this information. In fact Dekel and Herbsleb (2009) have already argued that this kind of knowledge should be “pushed” to developers to avoid them overlooking important information. In this article, we report on an empirical approach to automatically filter knowledge such as the above two sentences from larger documentation units, while retaining the possibility of displaying it in the context of the complete document.

Creating New Threads

In a programming task in Java with concurrency requirements, programmers typically create instances of a class implementing the `Runnable` interface and passing the instances to new `Thread` instances as shown below.

```
class Run implements Runnable ...
...
(new Thread(new Run())).start();
```

If the task is composed of multiple jobs demanding parallel execution, creating and starting a new `Thread` for each job is not efficient. To tackle some of these performance issues, Java 5.0 introduced the Executor framework in package `java.util.concurrent` (Austin, 2004). For the task above, it is more efficient to use a single instance of a class implementing the `Executor` interface, as shown below.

```
class AnExecutor implements Executor ...
...
Executor executor = new AnExecutor();
executor.execute(new Run());
executor.execute(new Run());
```

This recommendation of `Executor` over `Threads` is provided in the documentation of the `Executor` interface:

> *An* `Executor` *is normally used instead of explicitly creating threads*

but is buried in text that contains several different kinds of information, including code examples. The quote above is not highlighted in any way. How would a user discover this information? We cannot assume that users of the `Thread` class would be naturally inclined to read the entire reference documentation of the `Thread` class and of all the elements in the three Java concurrency packages (totaling about 91 000 words or over three hours of reading at an average speed). This example thus shows the need for information filtering (what information is indispensable or particularly valuable to users of class `Thread`?) and for information discovery (where is this information found in the reference documentation?).

Our research provides an initial approach for surfacing important statements like the one above, for their eventual recommendation to programmers. In this situation, we would detect that the programmer is using the `Thread` class, and return all text fragments previously classified as *indispensable* and *valuable* that are associated with `Thread`. Here fragments in the documentation of `Executor` become associated with the `Thread` class because they mention it.

3 Knowledge Items

Recommending knowledge from API documentation requires classifying the knowledge contained therein. In the first phase of our investigation, we manually studied the content of the Java 6 SE reference documentation (a.k.a "Javadocs") to elicit the properties that can help us distinguish information we can recommend.

3.1 Concepts and Terminology

In the JDK 6, reference documentation is mostly composed of text. In this project we focus on recommending text and ignore images and code. Textual documentation is composed of sentences, and a sentence or a group of sentences contains a unit of information, i.e., a self-contained message. For example, consider the following two units of information,[1]

> *If the limit array is not in ascending order, the results of formatting will be incorrect.*

and,

> *Note that the get and set methods operate on references. Therefore, one must be careful not to share references between* `ParameterBlocks` *when this is inappropriate.*

In the first case, a single sentence contains a unit of information, while in the second, there are two sentences. For uniformity, we refer to a sentence or a group of sentences that contains a unit of information as a **text segment**. We refer to the documentation explicitly **attached** to an API element, i.e., a class, an interface, an enum, a field, or a method, as a **documentation unit**. The documentation unit of a class or an interface thus only documents the class or the interface, and not its member fields or methods. A documentation unit consists of one or more text segments. If the unit of information present in a text segment is worth recommending as part of a programming task, we refer to the text segment as a **knowledge item** (KI). We say that knowledge items are **attached** to an API element if they are found in the documentation unit attached to the element. Alternatively, KIs can be **associated** with the element through various cross-referencing heuristics (e.g., if they mention the element, as described in Section 2).

3.2 Knowledge Items

We observed that in practice we can distinguish between two categories of knowledge items: *indispensable*, and *valuable*. Indispensable KIs are the pieces of information that the programmers cannot afford to ignore, such as, the *caveats*, or the *threats*, in using certain API elements. Valuable KIs offer useful but non-critical information. For example, a valuable KI could highlight the benefits of using one method over another to achieve a similar objective.

Some common high-level properties of both of these categories are that the pieces of information should be non-obvious for most programmers, and that they should have the potential to impact their decisions. For example, "*method X should not be called from method Y*" and "*it is not safe to call method X from method Y*" both instruct the programmer not to call method X from method Y; the former sentence does it explicitly and the latter implicitly. Text segments that merely state the property or the purpose of an API element do not involve programmer decisions, hence do not constitute a KI, for example, "*this enables the programmer to write code in a compact and easy style*".

We created an initial set of properties to characterize these two categories of KIs based on our own experience with API documentation (Robillard and DeLine, 2011), those of others in the field (Dekel and Herbsleb, 2009; Monperrus et al, 2011), and some established principles (Javadoc, 2001). We further expanded these properties using a grounded approach

[1] All quoted examples are taken from the reference documentation of the Java 6 SE APIs (http://docs.oracle.com/javase/6/docs/api/).

by closely studying the reference documentation of numerous API elements, followed by multiple refinement iterations.

We authored a *coding guide* intended to guide the manual classification of text segments. The coding guide describes the two knowledge categories, their properties with examples, and provides instructions on how to look for text segments that represent the two types of KIs in documentation units. In constructing our coding guide, we followed the best practices for the development of coding instruments according to the content analysis methodology (Neuendorf, 2002). The rest of this section describes our efforts in this respect.

Below, we provide a summary of the characteristics of the two knowledge categories as found in the coding guide.[2]

Indispensable Knowledge Items

Programmers who ignore this category of information would either encounter compilation or runtime errors or would be likely to introduce bugs. *Indispensable* KIs instruct programmers to perform important actions to accomplish basic objectives of an API element. An *indispensable* KI has properties related to one of the following subcategories:
Usage Directives specify non-optional directives or usage guidelines when using an API element. For example,

> *This method should only be called by a thread that is the owner of this object's monitor.*

Hard Constraints involve specific requirements. For example,

> *A valid port value is between 0 and 65535.*

Threats specify usage of certain protocols, whose violation would result in programming threats or errors. For example,

> *A* `CannotProceedException` *instance is not synchronized against concurrent multithreaded access. Multiple threads trying to access and modify* `CannotProceedException` *should lock the object.*

Valuable Knowledge Items

This is the type of KIs that conveys helpful and beneficial information. Programmers who ignore this category of information are likely to use the API sub-optimally (Kawrykow and Robillard, 2009), or spend an inordinate amount of time looking for information. A *valuable* KI has properties related to one of the subcategories below:
Alternative API elements recommend alternative API elements to accomplish the same objective but more efficiently. For example,

> *When using a capacity-restricted deque, it is generally preferable to use* `offerFirst`.

Dependent API elements recommend dependent API elements to help complete a task. For example, the `getFamily()` method in `Font` makes a reference to the `getName()` method using the sentence,

> *Use* `getName` *to get the logical name of the font.*

Such a piece of information, present in the documentation of a separate API element, is useful to keep track of dependencies between API elements. It can help programmers to figure

[2] An evaluation version is available on our project web page `http://swevo.cs.mcgill.ca/apireco/`

out API elements that are dependent on the one the programmer is working with (Duala-Ekoko and Robillard, 2011).
Improvement Options recommend actions that could lead to improvement in functionality or in non-functional properties such as performance. For example,

> *The implementor may, at his discretion, override one or more of the concrete methods if the default implementation is unsatisfactory for any reason, such as performance.*

Best Practices recommend practices that help make optimal use of the API element. For example,

> *While implementations are not required to throw an exception under these circumstances, they are encouraged to do so.*

Other Documentation

For the purpose of recommending documentation, the remaining content is of lesser importance, chiefly because it contains information that is unsurprising for programmers who have already selected the element, such as the basic objective of an API element (Cwalina and Abrams, 2008). We do not claim that this information has no value, but we estimate that the advantage of pushing it to developers during development tasks is uncertain. Our coding guide provides a detailed characterization of information that should be left out. Other documentation includes text segments that have some of the following properties (the complete list is found in our coding guide):
Obvious. A piece of information that is obvious from the name of the API element. Maalej and Robillard (2013) estimate that over 45% of the documentation units for methods and fields in the JDK 6 reference documentation contain at least one obvious text fragment (referred to as "non-information"). For example, for the method `getAudioClip(URL url, String name)`, the following line in its attached documentation unit contains the text:

> *Returns the AudioClip object specified by the URL and name arguments.*

Unsurprising. A piece of information that is unsurprising for most programmers (Cwalina and Abrams, 2008), for example, the summary sentence of API elements that provides a high-level objective or functionality of the element (Javadoc, 2001).
Predictable. A piece of information that is predictable based on the context, for example,

> *Exception* `SQLException` *is thrown if a database access error occurs.*

3.3 Reliability Assessment

For the purpose of manually coding text segments, the description of KI categories must be *reliable*. Reliability indicates with what consistency two independent coders (persons) would assign the same category to the same text segment (Neuendorf, 2002).

We measured the reliability of the coding guide by having two different *coders* independently identify indispensable and valuable KIs in randomly-selected documentation units. This evaluation was conducted by the first author and an external participant, a researcher who had recently joined our research group and had not previously collaborated with either authors or worked on this project. We performed the evaluation in three iterations involving, respectively, 77, 74, and 148 API elements, randomly selected from the Java 6 SE APIs. The non-uniformity in the number of API elements in each iteration is due to data collection

Table 1 Reliability assessment results. Iter = Iteration; DU = Documentation Units coded; T = Total number of text segments across all documentation units in the iteration; Ind = Number of text segments coded as Indispensable; Val = Number of text segments coded as Valuable; Rest = Number of other text segments; Dis = Number of instances of disagreement between coders; κ = Cohen's kappa value

Iter.	DUs	T	Ind.	Val.	Rest	Dis.	κ
1	77	201	11	25	152	13	**0.82**
2	74	133	14	21	85	13	**0.80**
3	148	244	19	43	155	27	**0.77**
Total	**299**	**578**	**44**	**89**	**392**	**53**	

times that were fixed in advance for each iteration. The coding task involved independently reading the assigned documentation units and identifying *indispensable* and *valuable* text segments. This phase of the research was iterative to allow improving the guide after each iteration. After each round, we studied every instance of disagreement and made additions and amendments to the coding guide as required.

We used Cohen's Kappa (κ) metric to measure the reliability between the two coders (Cohen, 1960). Unlike a simple percent agreement calculation, Cohen's Kappa takes into account the potential for agreement by chance. Kappa values are thus very conservative and values of 0.61-0.80 for κ are considered to indicate *substantial* agreement between two coders, and values of 0.81-1.00 are considered *almost perfect* (Landis and Koch, 1977).

In this phase of the research, aggregation of sentences into text segments was left up to the individual coders, so we could collect data to inform our automated segmentation technique (Section 4.1). There were occasional disagreements between the coders in deciding what constituted a text segment. In total 23 misalignments had to be manually reconciled to compute agreement values, i.e., only 4% of the total text segments eventually identified.

We reconciled misalignments using two heuristics. First, if a sentence was chosen by both the coders to form a text segment, but one of them included additional sentences either before or after the common sentence, we picked the larger text segment. Second, if one of the coders had selected two consecutive sentences as representing two different text segments each forming a different KI, while the other had selected both the sentences as part of a single KI, we chose the classification of the latter, and discarded the non-matching category selected by the former. The basis for employing these heuristics is that including more sentences as part of a text segment lowers the risk of breaking a unit of information.

Agreement measures between the two coders are presented in Table 1. The column *DUs* represents the total number of documentation units (API elements) coded in the corresponding iteration, and T is the total number of text segments across all the documentation units in the iteration.

We calculated κ for a 3-valued variable, i.e., each text segment could represent either an *indispensable* KI, a *valuable* KI, or neither of the two. In Table 1, the counts in the column *Ind.* indicates those text segments that were selected by *both* the coders as *indispensable* KIs, and similarly *Val.* for *valuable* KIs. The column *Rest* indicates the total text segments that were rejected by both the coders. The values in the column *Dis.* indicate the number of instances of disagreement.

The overall rate of disagreement was 9.2%, i.e., there were disagreements for 53 text segments out of the total 578. Out of the 53 disagreements, 7 were between Indispensable and Valuable classifications, 16 between Indispensable and neither, and 30 between Valuable and neither. The overall value of *0.80* for κ, however, indicates substantial agreement between the two coders. Although the value decreases slightly with each iteration, such small

differences at that level of agreement do not mean that our amendments were regressive; The larger final set simply contained documentation units with more ambiguous statements. With an overall κ in the $[0.77 - 0.82]$ range, we concluded that the guide was a reliable research instrument and used the version produced at that stage for the remainder of the project.

4 Automatic Detection of KIs

KIs are tedious to find manually, and there is a lot of documentation. The reference documentation for Java 6 SE alone consists of 206 packages, 3869 types, 28 724 methods, and 6158 fields. The reference documentation of all these API elements total 2 632 232 words in 194 204 sentences.[3] Obviously, some form of automated support is needed to extract knowledge items from existing documentation. As one of our primary contribution in this investigation, we developed a technique for finding KIs in large collections of documentation units.

In essence, the technique works by looking for pre-defined word patterns in documentation units. The patterns are discovered using a semi-automated technique, and then stored in a pattern database; they are not automatically learned through black-box text classification tools.

The justification for using specific patterns is two-fold. First, the cost of producing a large enough training set for KI classification based on Bayesian or Maximum Entropy techniques is prohibitive given that we have to classify sentences individually. Second, to a large extent word patterns for KIs are predictable and dictated by the type of information they encode. For example, we observed that KIs that represent *directives* usually have a modal verb with other supporting words, KIs that recommend alternate APIs use words like *recommend*, *advise*, or *prefer*, along with one or more terms referring to code elements (called *code words*), or groups of words like *use* and *instead*.

The existence of common linguistic patterns in reference documentation is not surprising given the existence of explicit style guides and description formats (Javadoc, 2001), as well as extensive examples from authoritative sources (the J2SE core packages).

From manually identified KIs, we produce patterns by extracting the words that are important to the knowledge conveyed in the sentence, and eliminating the words that are only supportive.

For instance, consider the *valuable* KI,

> *It may be more efficient to read the* `Pack200` *archive to a file and pass the* `File` *object, using the alternate method described below,*

For this KI, if we extract the words *may*, *efficient*, and the code term `File`, then an unseen text segment that would match this pattern is:

> *Consult your JDBC driver documentation to determine if it might be more efficient to use a version of* `updateBinaryStream` *which takes a length parameter,*

which has the words *might*, which we categorize as a modal synonym of *may*, *efficient*, and a code-like term `updateBinaryStream`.

In our technique, a *KI pattern* is simply a set of words, optionally including a special word that is a placeholder for code words, such as `File`. KI patterns do not contain duplicate

[3] http://docs.oracle.com/javase/6/docs/api/

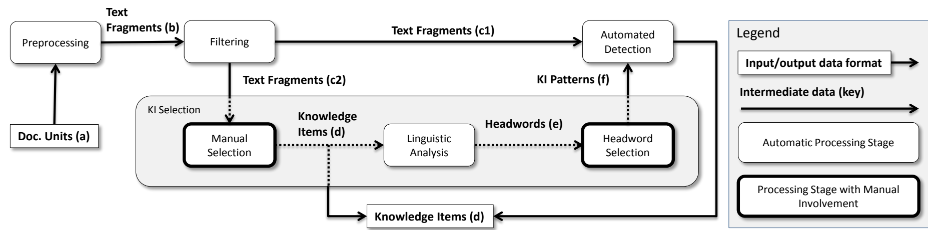

Fig. 1 The knowledge item identification process

words. As another example, the pattern {`should, wrap, <CW>`} would be derived from the following KI (where `<CW>` is the placeholder for code words):

> *If no such object exists, the list should be "wrapped" using the* `Collections.synchronizedList` *method.*

A sentence will match a KI pattern if it contains all the words in a pattern (or a synonym). We use the WordNet[4] dictionary to find synonyms.

Overview of the Approach

To proceed from raw documentation units to recommendations, our pattern-based approach requires the sequence of steps illustrated in Figure 1. The input to the approach is the raw text of all the documentation units in the reference documentation for a given set of APIs (in our case all of the Java 6 SE reference documentation (Javadocs), *a* in Figure 1), and the output is the set of all knowledge items estimated to be contained in this documentation corpus (*d* in the figure).

In the first stage, all documentation units are **preprocessed** to make them more amenable to automated analysis. This stage is described in Section 4.1.

In the second stage, we automatically eliminate text segments from the list of potential KIs using a fixed set of heuristics (Section 4.2).

With the remaining text segments, we do one of two things. A random sample is selected for **KI Selection** (*c2*), and the rest is reserved for automated detection (*c1*). The entire KI selection process is described in Section 4.3, and the automated detection in Sections 4.4 and 4.5.

Text segments selected for KI selection are first manually classified as described in Section 3. If they are determined to be KIs, they are stored in a final KI database (*d*), but also used to generate KI patterns. To generate patterns, we first use natural language processing techniques in a **linguistic analysis** stage to automatically discover and remove words unlikely to be important, and produce a list of "headwords" that are estimated to capture the essence of the KI. We then manually validate the headwords, producing a **KI Pattern** that is stored in a pattern database.

With a sufficient number of KI patterns, we can simply apply the patterns (*f*) to all remaining unclassified text segments (*c1*) to discover the KIs among them. These add to the manually-selected KIs to produce the final KI database containing all the KIs in the entire documentation corpus.

[4] http://wordnet.princeton.edu/

Once the KI database is available, we can use it as input to *Krec*, our documentation recommendation tool (Section 4.6).

4.1 Preprocessing

The *pre-processor* takes HTML documents containing documentation units as produced by the Javadoc tool and prepares them for further processing. It performs the following operations:

- It separates the documentation units for methods and fields from the Javadoc page of the type declaring them.
- It strips the HTML tags from the resulting documentation units, extracts the plain text, and identifies potential code words from the HTML `code` tags or based on additional heuristics. These heuristics include identifying camel-cased tokens, identifying tokens that are equivalent to the name of the API element associated with the documentation unit, etc.
- It converts the raw text into a list of sentences. We used the `PunktSentenceTokenizer` module in the Natural Language Toolkit[5] (NLTK) libraries to fragment text into sentences.
- It eliminates non-alphanumeric characters such as curly braces at the beginning or at the end of a sentence.
- It groups sentences that begin with a *conjunction*, e.g., *thus*, *hence*, *therefore*, etc., with their preceding sentence because they tend to represent a continuation of the same information.

The output of the pre-processor is a list of text segments.

4.2 Filtering

The filtering stage eliminates some of the text segments output by the pre-processor. This elimination is based on several heuristics:

- In Javadoc-style documentation units, the first sentence of “each member, class, interface, or package description” contains a high-level summary of what the element is supposed to do (Javadoc, 2001). Since our recommendations assume that the programmer has already selected an API element and has a basic understanding of what it should do, recommending a functionality summary from its documentation would be redundant as we would be recommending obvious information. We thus eliminate the first sentence from consideration.
- Documentation units often contain a mix of text and code examples. We eliminate text segments that only contain code blocks because we assume that the code would ideally be supporting information in text segments present either before or after the code. Hence the code block in isolation would neither contain *indispensable* nor *valuable* KIs.
- We also eliminate some of the independent clauses mentioned at the bottom of the documentation units for most API elements, especially methods. These include the clauses *Throws*, *See Also*, *Since*, *Specified by*, and the first sentence in *Returns*. The rationale behind each of these is present in our on-line appendix (see footnote 2).

[5] https://sites.google.com/site/naturallanguagetoolkit/Home

4.3 Knowledge Item Selection

We manually identify a set of KIs among filtered text segments (c2) to build a form of *training set* that is then used to generate KI patterns that will then be used to discover KIs in unseen documentation units. The procedure for transforming a KI into a KI pattern is almost fully automated, requiring only one simple word vetting phase at the end of the process. The basic idea for generating a pattern from a KI is to get rid of words that do not capture the essence of the KI. The first part of the procedure is fully automated and is represented by the sub-stage *Linguistic Analysis* in Figure 1.

In the linguistic analysis stage, KI sentences are tagged with their part of speech (POS) using the NLTK implementation of the `Treebank` tagger. Parts of speech describe the basic function of words (noun, verb, etc.). We further augmented the tagging technique to handle code words and to tag them distinctly. The output of this phase is a list of KIs consisting of words tagged with their POS. For instance, the first example of KI presented in this section would be tagged as follows:

```
It/PRP may/MD be/VB more/RBR efficient/JJ to/TO
read/VB the/DT Pack200/CW archive/JJ...
```

Where `PRP` is the tag for a personal pronoun, `MD` the one for a modal verb, `VB` the one for the base form of a verb, etc.[6] `CW` is our own tag, designed to represent code words.

We then pass the tagged KIs to a tool called a *chunker* to eliminate words that have a low probability of capturing the essence of a KI. Chunking (or shallow parsing) is a natural language processing technique that identifies different *phrases* present in the KI sentences. These phrases could be *noun phrases*, *verb phrases*, *adjective phrases*, etc. Each of these phrases consists of a *headword* accompanied by other *supporting words*. The headword is the word (or words) in the phrase with a POS tag that match the phrase. E.g., a noun phrase could have a noun headword accompanied by a determiner and an adjective.

Our approach is to consider that headwords are the most likely to capture the essence of a KI, and to disregard supporting words in a phrase, or words not associated with a phrase. The chunker relies on grammatical rules to create the different phrases. A good overview of the chunking technique can be found in a popular textbook (Jurafsky and Martin, 2008, §15.3).

Our initial set of rules included the standard English rules for the different phrase types. However, to use chunking to select essential words in KIs, it was not sufficient to use basic language processing tools: these needed to be specially engineered to properly handle the intricacies of software documentation. Code words, such as `addSource` in the KI "*a correct way to write the* `addSource` *function is to clone the source*," are treated as nouns. With observations from several manually-identified KIs, we also augmented the rules to extract words which may not be important in general usage of the English language, but are important in our context. For example, consider the KI,

> `JComponent` *subclasses must override this method like this:*

From this sentence, a rule for noun phrases extracts the words `JComponent` and *subclasses*, and a rule for verb phrase, the word *override*. Our additional rules further identifies `JComponent` as a code word term from its POS tag, and our customized *modal phrase* extracts the word *must*.

[6] The complete list can be found on our project web page (see footnote 2).

We customized the chunker to consider a modal phrase to be a modal verb accompanied optionally by verbs and adjectives. We categorized modal verbs into two categories: those that specify mandatory actions, i.e., *must*, *ought to*, *shall*, and *should*, and those that specify optional actions, i.e., *can*, *could*, *may*, *might*, *will*, and *would*.

Continuing with our example, the chunker would assemble words into the following phrases (headwords are in italics, code words are indicated with an asterisk):

```
It/PRP
(may/MD) Modal Phrase
be/VB
(more/RBR efficient/JJ) Adjective Phrase
(to/TO read/VB) Verb Phrase
(the/DT Pack200*/CW archive/JJ) Noun Phrase
...
```

From each phrase detected in a KI, all supporting words (non-headwords) are automatically eliminated. The remaining list of headwords is then produced as the output of the automated linguistic analysis stage. The final stage (Headword Selection) requires a human to look at the list of headwords and eliminate any word that does not usefully capture the essence of the KI.

In our example, the output of the linguistic analysis stage would produce the following list of headwords with their accompanying phrase (stars indicate code words):

```
Modal Phrase: may
Adjective Phrase: efficient
Noun Phrase: Pack200*
Verb Phrase: read
```

From which we extract the pattern {`May, Efficient, <CW>`}.

Finally, we note that because each of the patterns relates to a KI, and each KI is associated with a category, the patterns are likewise associated with a category (i.e., *indispensable* or *valuable*).

4.4 Training Set Construction

To proceed with the automated detection stage of our approach, we needed a sufficiently large collection of KI patterns generated from manually-identified KIs. The challenges for this stage were that *a)* each pattern involves the manual inspection of a KI, and *b)* our collection of patterns must be diverse enough to capture a sufficiently large number of KIs to be able to produce recommendations in a majority of programming contexts.

Our basic strategy for meeting these inter-related challenges was to build the training set incrementally through a combination of automated detection and manual validation. In addition to speeding up the process of building the pattern database, this process allowed us to estimate the performance of our automated KI discovery process. We summarize the process as follows:

1. Manually generate a seed training set of validated KIs, and generate patterns from them. Our seed training set consisted of all the KIs discovered as part of the development of the classification scheme (Section 3), plus those discovered in an additional 100 randomly-sampled documentation units.

2. Generate an *intermediate test set* by randomly sampling 20 unseen documentation units.
3. Apply all known patterns to this intermediate test set. Identify false negatives and false positives (missing KIs).
4. Discard the false positives and add the missing KIs identified in the previous step to the training set. Generate patterns from them. Add the generated patterns to the pattern database.
5. Randomly sample 100 additional unseen documentation units. Manually identify KIs in them, and add the corresponding patterns to the pattern database.
6. Go to step 2.

After six iterations, we had inspected 899 documentation units in the training sets and 120 in the intermediate test sets. For the total of these 1019 documentation units, which is about 2.6% of the total in the Java 6 SE SDK, we collected 556 *unique* KIs, and 361 *unique* patterns; 142 *indispensable* and 219 *valuable*.

Across our six test iterations on 120 documentation units, we correctly discovered 38 KIs, incorrectly produced five spurious KIs (false positives), and omitted 42 KIs (false negatives). This gave us a preliminary precision of [0.82–1.00] and a recall of [0.30–0.44] depending on the test iteration.

Based on our experience so far, we estimate that further increasing the size of the pattern database in the same way will lead to higher recall and lower precision. Given the massive investment required to produce the current database, we opted to apply the pattern database at this point in the process.

We note that the precision and recall numbers provided here are the intermediate results that we used to determine when to stop adding to our training set. The results of the end-to-end evaluation are presented in Section 5.

4.5 KI Database Generation

From the remaining 97.4% of the documentation units remaining unseen, 57.2% of them (21 700) had one or more sentences left after the filtering stage. We then applied the 361 patterns to automatically identify potential KIs. The automated detector was able to detect one or more KIs in 8396 (38.7%) of them. We stored the automatically detected KIs in a final *KI database*.

Table 2 presents the details of the KIs generated by the automated detector and those manually vetted. The column *Total* shows the number of documentation units associated with each API element type, and the column *With KI* reports how many of those contain at least one KI. Out of the 8396 elements for which there is one or more KI in the corpus, 75% are methods, 20% are types, and the rest are fields.

The rest of the table reports on the total number of KIs in our database, listed by documentation unit type, KI type (indispensable or valuable) and origin (*Manual*ly or *Auto*matically detected). Out of all the automatically detected KIs, 80% belongs to the documentation unit of methods.

The most effective pattern was `{<CW>, must, pass}`, where `<CW>` is a placeholder for a code word; it matched 1072 KIs across all the documentation, which is 7.6% of the total KIs automatically detected. 265 out of the total 361 patterns matched one or more KIs (in addition to the seed KI). The average instances of a match per pattern was 52.

Table 2 Knowledge item corpus. The column *Total* shows the number of documentation units associated with each API element type. The column *With KI* reports how many of those contain at least one KI. The columns under *KIs* report the total number of KIs in our database, listed by documentation unit type, KI type (Ind = Indispensable; Val = Valuable), and origin (Manually or Automatically detected)

	Doc. Units		**KIs**			
	Total	**With KI**	**Manual**		**Auto**	
			Ind.	**Val.**	**Ind.**	**Val.**
Types	3869	1685	91	148	797	1817
Methods	28724	6301	120	168	3514	7217
Fields	6158	410	10	19	190	506
Total	38751	8396	221	335	4501	9540

BlockingDequeExample.java

```
package util;

import java.util.concurrent.BlockingDeque;

public class BlockingDequeExample
{
    public static void main(String[] args)
    {
        BlockingDeque<String> deque = new LinkedBlockingDeque<String>();

        deque.addFirst("first");

        deque.addLast("last");

        System.out.println(deque.size());
    }
}
```

(A) Input code

Knowledge Recommender

(B) Recommendations of KIs corresponding to the elements in the code

[V] java.util.concurrent.BlockingDeque.addFirst(E)::When using a capacity-restricted deque, it is generally preferable to use offerFirst
[V] java.util.concurrent.BlockingDeque.addLast(E)::When using a capacity-restricted deque, it is generally preferable to use offerLast

Fig. 2 Krec in action.

4.6 Knowledge Recommender

We developed an Eclipse plug-in called *Krec* (Knowledge Recommender), that takes as input a Java file or a block of Java code, and recommends KIs *associated* with the API elements in the code. *Krec* uses as input the KI database generated as described in this section. *Krec* also recommends KIs attached to elements not present in the input code if they mention one or more of the elements in the code. E.g., when using `Thread` in the input code, it recommends a KI from the Javadoc of `Executor`, because this Javadoc mentions `Thread`.

Figure 2 shows a screen shot of *Krec*; the programmer looks for recommendations for lines of code in the program by selecting the code block and initiating *Krec*. *Krec* identifies the API elements in the code and recommends KIs associated with the elements, if it finds them in the corpus. If a user selects a KI from the list, a browser displays it with its full context.

Although fully-functional, *Krec* is a proof-of-concept prototype that we used to facilitate data analysis and experiment with the approach. This paper focuses on solving the data extraction challenge underlying API documentation recommendation. For this reason, we do not claim that Krec could be put in operation without additional engineering efforts. In particular, a fully-functional recommender system requires support for *user profiling*, in particular to model the knowledge of the developer to avoid recommending knowledge that the user already knows. A basic solution, illustrated by the work on CodeBroker, would be

Table 3 Target open source systems

System	Version	Purpose
ArgoUML	0.34	UML modelling application
FreeMind	0.9.0	Mind mapping application
Hadoop	1.0.3	Distributed processing framework
Hibernate	4.1.4	Object-relational mapping framework
JDT	3.7.2	Tools for Java IDE
JEdit	4.5.2	Text editor
Joda Time	2.1	Java library for date and time
JUnit	4.11	Testing framework
Tomcat	7.0.28	Web server
XStream	1.4.2	Serialize Java objects to XML

to maintain a list of KIs assumed to be known by the user (Ye and Fischer, 2002). Fully solving this problem requires the consideration of a very large number of factors and largely exceeds the scope of this work (Ying and Robillard, 2014).

5 Evaluation

We conducted three studies to evaluate the potential of pattern-based detection and recommendation of knowledge items from API reference documentation. In a first study (Section 5.1) we generated recommendations for code from a sample of ten open-source systems to estimate the precision and recall of the patterns in our database. In a second study (Section 5.2), we applied Krec to well-documented code examples from the literature to verify that our approach could identify important knowledge associated with the examples. In a third study (Section 5.3), we involved ten independent participants to assess the potential usefulness of KIs detected with our approach.

5.1 Production Code

We generated recommendations for code from ten systems that use the JDK APIs in different ways (Table 3). The systems in this table represent different domains and vary significantly in their size. These systems also differ in their use of the JDK APIs. For example, Joda Time uses only basic APIs such as the collection classes, whereas JEdit relies on AWT and XStream uses the reflection APIs.

For each system, we randomly selected 20 method definitions from the population of methods consisting of more than five lines of code and using at least one JDK API element. Additionally, we *stratified* our sample and ensured that each method was selected from a different package. The size of the test sample is limited by the high effort in manual investigation required to compute the precision and recall metrics used to assess the results.

Table 4 shows the distribution of API elements in the sample. The column *Total LoC* indicates the total lines of code across the 20 methods in each system, and *Total APIs* indicates the sum of the occurrence of all the *non-trivial* JDK API elements; we obtained the *non-trivial* elements by filtering out pervasive elements such as `PrintStream`, `System`, etc. The complete list is in our on-line appendix (see footnote 2). *Distinct APIs* indicates the number of API elements in the sampled method definitions in each system with duplicate occurrences of elements removed. *Unique APIs* indicates the count of the API elements that

Table 4 Description of the evaluation sample. Total LoC = total lines of code across the 20 methods; Total APIs = sum of the occurrence of all the non-trivial JDK API elements (defined in the text); Distinct APIs = number of API elements in the sampled methods with duplicates removed; Unique APIs = Number of API elements only present in the sample for that system

System	Total	Total	Distinct APIs		Unique APIs	
	LoC	APIs	Count	(%)	Count	(%)
ArgoUML	237	119	99	83.2	61	51.3
FreeMind	241	141	125	88.7	94	66.7
Hadoop	222	147	126	85.6	66	44.9
Hibernate	181	148	92	62.2	41	27.8
JDT	307	104	75	72.1	26	25.0
JEdit	313	157	138	87.9	77	49.0
Joda-Time	323	111	86	77.5	44	39.6
JUnit	201	85	63	74.1	19	22.4
Tomcat	217	111	82	73.9	42	37.8
XStream	245	154	101	65.6	36	23.4
Total	2487	1277	987	77.3	506	40.0

Table 5 Evaluation results—production code. KI Rec = Total number of recommended KIs of each type; Prec = Precision; KI Missed = Total number of KIs of each type present in the corresponding documentation unit but not recommended; Rec = Recall; Extra = Percentage of text segments in the documentation units for all the elements in a method declaration that are not KIs; #X number of methods with KIs recommended from documentation units other that the ones attached to API elements used in the method

	KI Rec.		Prec.	KI Missed		Rec.	Extra	#X
	Ind.	Val.	(%)	Ind.	Val.	(%)	(%)	
ArgoUML	26	83	87.6	6	26	77.3	85.4	5
FreeMind	13	48	93.8	5	14	76.3	90.3	6
Hadoop	29	59	93.2	9	30	69.3	87.5	6
Hibernate	35	69	87.2	2	24	78.8	87.4	2
JDT	23	51	83.8	3	19	77.1	86.2	1
JEdit	14	50	98.1	9	33	60.4	91.4	2
Joda Time	25	66	90.2	23	53	54.5	82.0	3
JUnit	21	46	89.1	12	34	59.3	82.9	3
Tomcat	19	38	87.7	10	32	57.6	88.5	3
XStream	39	89	90.6	9	34	74.9	82.5	1
Total	244	599	90.1	88	299	68.6	86.4	32

are only present in the sampled method definitions of that system. Hence, on average 50.6 elements are unique to each system in the sample.

We input the method definitions to *Krec* and measured four metrics: *precision*, *recall*, *extraneous information* (Extra) and *number of methods with cross-references* (#X). Precision is the ratio of recommendations that are actual KIs over all recommendations. Recall is the ratio of recommendations that are KIs over all KIs found in the documentation units attached to API elements in respective method declarations. The extraneous information measure is the percentage of text segments in the documentation units for all the elements in a method declaration that are not KIs. This measure illustrates to what extent our approach addresses the information filtering problem. The number of methods with cross-references shows the number of methods in our sample for which at least one of the recommended KIs came from a documentation unit not attached with any API element in the method definition (e.g., the `Executor` example in Section 2). This measure illustrates to what extent our approach addresses the information discovery problem. Table 5 shows the results.

On average, *Krec* was able to recommend KIs with 90% precision. It identified one *indispensable* KI for every 5.2 API elements and 10.2 lines of code, and one *valuable* KI for every 2.1 elements and 4.2 lines of code. These results translate into reasonable usability projections, since the number of recommendations is both manageable (one for every two elements), and the precision is high. Simple user interface features have the potential to further decrease the number of spurious recommendations generated (we discussed this feature in Section 4.6).

Krec achieved an overall recall of 68.6%. On average, it missed 1 *indispensable* KI for every 2.7 that it found, and 1 *valuable* KI for every 2 that it found. We observe that Joda-Time has a lower recall compared to other systems because it uses `Date`, `Calendar`, and `Locale` APIs that apparently have unique sets of information in their associated documentation unit, and were not part of the training set. The implications of the recall measures are straightforward: developers should not rely blindly on being recommended all the information they need. Recommender systems can function as a mean to facilitate information discovery in reference documentation, but should not be seen as a means to replace it.

In terms of *extraneous information*, we note that, on average, 13.6% of the crucial documentation contained all the detected KIs, meaning that the approach helps filter out about 85% of text segments which may be less relevant in our intended usage scenario.

Finally, the number of methods with cross-references shows that in $32/200 = 16\%$ of recommendation contexts (method declarations), our approach recommended information not found in the documentation units attached to the API elements used in the method. Although this number illustrates the basic ability of the approach to support information discovery, it is still modest and we are considering new ways to link KIs with API elements. One promising avenue is to also recommend KIs found in documentation units listed under the “See Also” rubric for a given API element. This idea is only one illustration of the numerous ways possible to increase information discoverability using a recommender system.

The results in Table 5 illustrate the average performance of our approach, and aggregate multiple occurrences of the same recommendation. Since the recommendations are context independent, they will always be the same for a given element. In other words, if the recommendations for a given element, such as `Thread`, are very good, the results will (indirectly) be a function of the number of references to `Thread` in our sample. Hence, the results in Table 5 paint a useful picture of the expected average performance in practice, but do not allow us to account for the frequency of individual elements.

To control for this factor, we studied the performance of *Krec* on individual elements (Table 6).

Table 6 Automatically-generated KIs per element. Total APIs = Number of API elements referenced in the sample; Distinct APIs = Set of API element referenced in the sample; APIs with auto gen. KIs = Number of the distinct API element for which at least one KI was generated automatically; Auto Gen KIs = Number of KIs of each type generated; True KIs = KIs manually verified as correct; Missed KIs = KIs found in the corresponding documentation units that were not detected; P_{micro} = Micro-averaged precision; R_{micro} = Micro-averaged recall

Total	Distinct	APIs with	Auto Gen. KIs		True KIs		Missed KIs		P_{micro}	R_{micro}
APIs	APIs	auto gen. KIs	Ind.	Val.	Ind.	Val.	Ind.	Val.	(%)	(%)
1277	660	186	48	167	32	131	36	128	75.8	49.8

Across the 200 method definitions in our sample, there was a combined usage of 1277 API elements. Out of this total, 660 were distinct. From these 660 elements, 31 were part

of the manually-validated training set, and *Krec* was able to automatically generate one or more KIs for 186 of the remaining elements. For these 186 elements, *Krec* automatically identified 215 KIs, with a micro-averaged precision of 75.8%. With micro-averaging, the decisions are accumulated across elements they refer to, and the measures are computed on the accumulated contingency table (Manning et al, 2008, p.261). The values from the individual elements were computed by reading their attached documentation unit and analyzing each text segment for the presence of KIs. This is lower than the average precision per system reported in Table 5 because Table 5 contains manually validated KIs and multiple occurrences of some KIs. The macro-averaged precision, i.e., the average of the precision of KIs recommended per element, did not apply for our case, because the number of KIs per element is low; for those with a KI in their associated documentation unit, the number of KIs ranges from 1 to a maximum of 8, with more than 50% of the cases having only one KI. Because precision (and recall) suffer from high variance (e.g., 0 or 100 on documentation units with one KI) on such small samples, we report only the micro-averaged value.

For the 660 distinct elements across all the systems, *Krec* missed 164 KIs for a recall of 49.8%. On examination, all KIs belonged to 280 elements out of the total 660, which provided a coverage of 42.4%.

The main reason for a miss is the lack of a matching instance. The KIs that are unique to an API element are difficult to extract automatically without a precise pattern. This is true for cases where the sentences representing a KI do not have well-defined headwords. For example, the following KI, that indicates a probable threat,

This function may cause the component's opaque property to change.

contains a specific piece of information, without distinct headwords or code-like terms, hence the probability of it matching a pattern from other KIs is low. We noticed that most of such cases are true with short sentences. As a solution to this, it would be appropriate to create a stratified sample of short sentences in the training sets.

5.2 Targeted Examples

We also evaluated whether our current KI database for Java SE 6 would be able to make useful recommendations in cases where we know a recommendation is necessary. For this purpose we used a collection of code examples taken from the book Effective Java (Bloch, 2008). This book presents a number of code examples with documented problems and the accompanying solution. Specifically, Effective Java (EJ) contains 78 rules intended to make the most effective use of the Java programming language (Bloch, 2008). We broadly categorized these 78 rules into two types:

1. those that recommend general *programming best practices*, e.g., "*never do anything time-critical in a finalizer*" (Bloch, 2008, p. 27).
2. those that recommend means of *effective usage of the fundamental Java SE libraries*, e.g., "*always override* `toString()`" (Bloch, 2008, p. 51). The fundamental libraries include `java.lang`, `java.util`, `java.util.concurrent`, and `java.io`.

We manually separated the 78 rules into these two types; if the rule involved an API element, i.e., ways of *using* or *not using* an API element, we put it in the second category, and we put all the other rules, including the ambiguous ones, in the first. For all the rules of the second type, we extracted the associated sample code (Bloch, 2008), and input it to *Krec*.

Table 7 Evaluation results—targeted examples. KI Rec = Number of KIs recommended for the code example; KI Missed = Number of KIs present in documentation units associated with API elements in the code example that were not recommended; Expected = Whether the author's main recommendation associated with the code example was also found by *Krec*

	KI Rec.		KI Missed		Expected
Effective Java Items	**Ind.**	**Val.**	**Ind.**	**Val.**	
5 Sum.java	0	2	0	0	NA
8 CaseInsensitiveString.java	0	3	0	2	Yes
9 PhoneNumber.java	6	15	0	2	Yes
10 PhoneNumber.java	8	18	0	4	Yes
11 Stack.java	1	7	0	2	Yes
12 WordList.java	0	3	1	3	NA
14 Complex.java	3	6	5	4	NA
29 PrintAnnotation.java	1	2	0	3	No
36 Bigram.java	2	1	0	6	No
47 RandomBug.java	2	4	1	4	Yes
49 BrokenComparator.java	3	5	1	2	Yes

Table 7 shows the evaluation results on the sample code associated with the EJ rules; the Java files are downloaded[7] from the EJ web location (Bloch, 2008). For these examples, we studied whether an obvious, expected recommendation was produced by *Krec*. Out of the 11 EJ examples having to do with API usage, eight have the suggested solution buried in the documentation unit. Out of these eight, *Krec* was able to find the corresponding recommendations for six of them.

The 2 cases where it was not able to identify the expected recommendations are:

1. Item 29, which relates to the use of the method `asSubClass` in `Class` to safely cast an object (Bloch, 2008, p. 146). The equivalent fact in the documentation unit of `asSubClass` is presented as a *purpose*, which we do not capture in KIs.
2. Item 36, which uses an overloaded `equals` instead of `overridding` it (Bloch, 2008, p. 177). As a result, *Krec* could not identify the constraint associated with the original `equals` method.

5.3 Perceived Usefulness of the Knowledge Items

The evaluation described in the previous subsections focuses on the quality and generalizability of the patterns for detecting knowledge items. How useful the knowledge items can be to developers is a separate question which we investigated through a survey of ten participants with software development experience. The study involved asking each participant to rate 30 knowledge items detected with our approach.

Sampling and Participants

As part of this research we created a corpus of 14 597 automatically-generated knowledge items (summing the four rightmost values at the bottom of Table 2). For practical reasons it is not possible to reliably evaluate the usefulness of all these knowledge items. Instead we must rely on a sample of items to evaluate. Given the resources to evaluate a fixed number N of KIs, the main experimental design question we faced was how to sample KIs across

[7] http://java.sun.com/docs/books/effective/effective2.zip

all types listed in Table 2 (Indispensable vs. Valuable, Types vs. Methods and/or Fields, etc.). Sampling broadly (across all classes) supports weak claims across a wide population, whereas sampling narrowly (within only a single type) supports reliable claims across a small population. Given this tradeoff and resources available to evaluate an estimated $N = 250$ KIs, we opted to sample within the population of automatically-generated KIs for types that were associated with the label "Indispensable" (a population of 797 instances). These numbers produce a binomial confidence interval at the 95% level of ± 5.14 for a proportion of 50%. In other words, if 50% of respondent answer "yes" to a yes-no question about a KI, the true proportion lies within the interval $[44.86\%, 55.14\%]$, 95% of the time. Our choice of the class "Indispensable KIs for types" as the reference population was based on our intuition that these would be the most useful KIs. Because this work is our first experience with this approach, we were interested in collecting initial evidence of the usefulness of the approach. We removed KIs associated with the packages `org.*`, such as `org.omg.*` and `org.xml.*`, leaving a total of 749 KIs in our population, because these APIs are not in common use and unlikely to be familiar to software developers we could recruit for this study.

We recruited ten participants to complete the evaluation of the sampled KIs. Five of the participants were from industry (their experience in industry ranged from 1 to 5 years) and five were from academia (three PhD students and two Masters students). The experience of the ten participants in Java ranged from 1 to 8 years with a mean of 3.5 years.

To measure how participants agreed on the value of a KI, we randomly selected 50 KIs from our sample and assigned them to exactly two random participants, which gave us 100 ratings. Then we selected another random 200 KIs and assigned them to exactly one participant, giving us another 200 ratings. This design required a total of 300 ratings, with each participant rating 30 KIs.

Survey Instrument and Tool

We implemented the survey as a web application that presented each KI and its associated type one by one to a participant, and asked the participant to choose the statement that best described the KI from the following four options:

1. *This information would help a developer who does not already know it to better use the above API type in general.*
2. *This information looks relevant to learning how to use the API type but it requires more context to be fully usable: It might be worth looking up.*
3. *This information would not be useful or usable in most programming situations.*
4. *I am not able to properly answer this question.*

Figure 3 shows a screenshot of the survey application. The tool allowed the users to complete the survey in multiple sittings but it did not allow them to change their responses.

Results

Option 1, which indicates a useful KI, was selected 172 times out of 300 ratings (57%, sd=21%). Option 2, which indicates some potential for usefulness, was selected in 90 cases (30% sd=13%). Finally, option 3, indicating a false positive (useless KI) was selected 32 times (11% sd=10%). In only six cases (2%) were the participants unable to answer (option 4).

Fig. 3 The Web Application Survey on a Test User

We analyzed the agreement between participants and found that participants agreed on exactly the same rating for 31 out of the 50 overlapping KIs (62%). Most of the disagreements (12/19) were caused by differences of opinion over whether a KI should be rated as useful (option 1) or potentially useful (option 2). Six of the disagreements were between potentially useful (option 2) and not useful (option 3). The other disagreement involved a participant rating the KI with option 4 (unable to answer). There were no cases were participants disagreed between useful and not useful.

We noticed that in a majority of the cases, participants classified a KI as potentially useful (option 2) if the KI contained code words that were not obviously related to the type in question, such as in the following case for type `javax.xml.validation.ValidatorHandler`:

> *Applications must ensure that* `ValidatorHandler`*'s* `ContentHandler.startPrefixMapping(String,String)` *and* `ContentHandler.endPrefixMapping(String)` *are invoked properly. Similarly, the user-specified* `ContentHandler` *will receive startPrefixMapping/endPrefixMapping events.*

Interestingly, one common characteristic for a number of the KIs classified as not useful (option 3) was their association with concurrent programming. For example, the following KI in the documentation unit of `java.util.LinkedList<E>`:

> *If no such object exists, the list should be "wrapped" using the* `Collections.synchronizedList` *method.*

Or, the following KI in the documentation of, among others, `java.security.cert.PKIXCertPathChecker`:

> *Unless otherwise specified the methods defined in this class are not thread-safe.*

For cases were participants were unable to decide (option 4), we surmise that the KIs required an excessive amount of context to interpret. For example, the documentation unit of `java.security.SecurityPermission` contains descriptions of multiple permission types, including the following:

setPolicy: Setting of the system-wide security policy (specifically, the Policy object) - Granting this permission is extremely dangerous, as malicious code may grant itself all the necessary permissions it needs to successfully mount an attack on the system.

Interpreting this statement would require understanding other permission types in the class.

If we take the pessimistic view and only consider KIs rated with option 1 to be valuable recommendations, we can conclude that KIs tagged as Indispensable that are recommended for API types would be valuable to developers in $57.3 \pm 5.01\%$ of cases, 95% of the time. If we include potentially-relevant KIs (option 2), this proportion rises to $87.3 \pm 3.37\%$, 95% of the time. Although this study does not allow us to make usability claims about other classes of KIs, they provide evidence of usefulness that motivate further effort to deploy the tool to recommend at least the indispensable KIs for APIs types. In the case of success in a production environment, other classes of KIs could then be incrementally added to the system and assessed.

5.4 Threats to Validity

The process of identifying KIs involves manual intervention at two points: identifying the *indispensable* and the *valuable* KIs in the training set, and identifying the essential headwords from the list output by the automated filter. Though we observed high agreement with an external participant for the classification, both tasks involve personal judgment; a liberal selection will increase recall and decrease precision, and vice-versa. The coding guide assumes that the documentation follows the Javadoc principles (Javadoc, 2001). If a documentation author deviates from the convention, the training set would need to be expanded accordingly. Given that our approach depends on the writing style of documentation, we emphasize that our quantitative results are a function of the style employed for the reference documentation of the JDK 6. Although there is much variety in style and quality within the JDK 6, this corpus cannot be expected to be representative of all existing reference documentation. Although we evaluated the automated detection of KIs on varying domains, KIs with a strong domain flavor are less likely to have matching patterns, leading to low recall.

In the case of our evaluation of the perceived usefulness of the KIs, we note that the quantitative results only generalize to indispensable KIs for types. Additionally, to minimize the load on participants we did not ask them to rate their expertise with the types associated with the KIs they evaluated. It may be possible that perceived usefulness of a KI is related to pre-existing knowledge (or absence of knowledge) about the type. However, during the study participants had the opportunity to consult the original API documentation to clarify any question if necessary. In general there is a trade-off between the amount of contextual information collected from the participants and the total number of KIs we could evaluate: the higher the number of data points, the less the aggregated results will be sensitive to internal validity threats. Finally, we point out that the evaluation of the usefulness of KIs is based on a subjective assessment from the survey respondents. Although the number of responses eliminates the threat of an individual bias, it is nevertheless possible that the responses may be affected by a collective bias.

6 Related Work

This work builds on studies of API documentation and a vast body of work on the application of natural language processing in software engineering. We highlight representative work in both areas.

API Documentation

The impact of documentation on the usability of APIs has been an area of active research (Stylos et al, 2009; Brandt et al, 2009; Robillard, 2009; Dekel and Herbsleb, 2009; Robillard and DeLine, 2011). Researchers have made proposals to make API documentation easily accessible and understandable to programmers.

Nykaza et al (2002) found that programmers used reference documentation only when they fail to get enough information from other possible sources. This could be due to the nature of the presentation or the content of the documentation. Among other findings, Nykaza et al. identified the importance of an overview section in API documentation, and Jeong et al (2009) identified the importance of explaining starting points to increase the quality of the documentation. We developed techniques to distinguish parts in the reference documentation that are irrelevant to programming situations from the parts that are relevant.

Dekel and Herbsleb (2009) worked on highlighting directives present in Javadocs of several major APIs. Their tool, *eMoose*, can push directives from documentation into the foreground to apprise the programmer of their presence. *eMoose*, however, relies on tags in the documentation to identify directives. Even though their approach provides several helpful directives to the API user, it puts an overhead on the API developers and contributors to include such tags in the documentation. Also, it would be difficult to identify directives in existing documentation that are void of such tags. Their work was one of the sources of inspiration that stimulated our interest in developing automated techniques for finding knowledge to push to developers. In addition to just directives, we automatically identify other important forms of information, for example, alternative API recommendations.

Monperrus et al (2011) performed an extensive empirical study and identified all possible directives in three large Java projects: JDK, JFace, and Commons Collections. We, however, claim that not all directives are equally important, hence we identify only those that require the programmer to make a decision. Such directives provide an immediate API usability improvement to the programmer.

Maalej and Robillard (2013) categorized API reference documentation into twelve patterns of knowledge. They did an elaborate study to summarize knowledge in API documentation into these patterns and then involved 17 participants over several weeks to identify the patterns in the JDK and the .NET API documentation. In contrast to this work, which focused on determining the quality of API reference documentation, we focus on the automated extraction of information in the documentation that can be effectively recommended during a task.

Stylos et al (2009) proposed *Jadeite*, which studies source code and statistically provides recommendations to the programmers on the most used classes, constructors, methods, and objects. These specifications helped detect bugs which were introduced due to developers using APIs for purposes that were not intended by the API. Kim et al (2009) proposed *eXoaDocs*, which uses code snippets, mined from search engines, to improve documentation, by integrating the code with the text. In contrast to *Jadeite* and *eXoaDocs*, which need

external input to improve the documentation, our work is focused on making the existing information in the reference documentation more accessible by extracting the relevant and eliminating the irrelevant pieces of information.

NLP in Software Engineering

Natural Language Processing has been used in a wide variety of applications intended to assist software engineering tasks. Most applications of NLP involve preprocessing and linguistic analysis similar to what is described in Section 4. However, the research challenge in designing NLP systems to assist software engineering is one of *domain adaptation*: to determine how to process, adapt, and combine data sources and text analysis methods to effectively support a software engineering task. The following discussion illustrates the major classes of applications that can be supported with NLP.

Software requirements are to a large extent expressed in textual form, so natural language processing can assist with a wide variety of tasks. For examples, NLP techniques have been used to support the discovery of new requirements by mining the text of on-line forums (Hariri et al, 2014). At the other end of the spectrum, NLP techniques have also been used to identify missing objects and actions in requirements documents (Kof, 2007), as well as to quantitatively estimate the quality of existing requirements (Fantechi et al, 2002).

Natural language processing is also at the core of techniques intended to discover undocumented software specifications. For example, Arnout and Meyer (2003) proposed a technique to infer invariants, like preconditions and postconditions, from the documentation; Tan et al (2007) proposed *iComment*, which extracts specifications from comments in source files; Work has also been done by Xie and colleagues to infer various types of specifications from natural language text, including method call protocols (Pandita et al, 2012), resource specifications (Zhong et al, 2009), and security policies (Xiao et al, 2012).

Finally, NLP techniques are invaluable for helping users sift through the massive amount of information available to software developers, be it in documentation or in the source code itself. For example, Pagano and Maalej (2011) used an unsupervised text clustering technique called latent Dirichlet allocation (LDA) to classify blog posts to infer the nature of their contents. Henß et al (2012) also used LDA, but to semi-automatically build summaries in the form of “Frequently Asked Question” (FAQ) documents. Our work explores a different dimension of language analysis in software engineering by both focusing on a different linguistic register (systematic, official documentation), a different approach (semi-supervised pattern discovery), and a different application (recommendation as opposed to summarization). In the area of software comprehension, applications of NLP include the work of Shepherd et al (2007), who used NLP to locate and comprehend specific concerns in source code, and that of Panichella et al (2012), who mined bug reports and developer mailing lists to discover descriptions that can suitably explain methods.

7 Conclusion

We presented an approach to detect *knowledge items (KIs)* in API reference documentation that may be *indispensable* or *valuable* to a programmer who has already selected an API element to be part of a solution. Our approach relies on the automated detection of knowledge items based on word patterns that capture the essence of the knowledge being conveyed.

With a training set of 556 knowledge items extracted from a corpus of over 1000 documentation units for elements from the Java 6 SE API, we produced a database of over 14 500 knowledge items that can be *recommended* during programming tasks when a programmer uses an API element associated with a KI.

We evaluated the approach using an Eclipse plug-in, *Krec*, on 200 method definitions extracted from 10 varied open source systems. With our KI database, under estimated operating conditions we could issue recommendations with, on average, 90% precision and 69% recall. At this level of performance, the approach was able to filter 86% of the documentation text attached with the corresponding source code, and recommended knowledge found in non-obvious documentation locations in 16% of the 200 recommendation contexts. We also verified that obvious recommendations from textbook examples could be recommended, with a success rate of 6 out of 8 cases. Finally, an evaluation with ten independent assessors showed that Indispensable KIs associated with JDK SE 6 types were expected to be useful to software developers at least between 52.3% and 62.3% of cases, 95% of the time.

Naturally, building an industrial-strength recommender for API documentation will requires much additional performance optimization and feature development work. However, our results provide us with the evidence that using word patterns is a cost-effective approach for finding important knowledge in reference documentation, thus opening the door to a practical way to support recommending API documentation.

Acknowledgements We thank Peter Rigby for participating in the coding and Walid Maalej, Christoph Treude, and Annie Ying for comments on the paper. This work was supported by NSERC.

References

Arnout K, Meyer B (2003) Uncovering hidden contracts: The .net example. Computer 36(11):48–55

Austin C (2004) J2SE 5.0 in a nutshell. Sun Developer Network Article, `http://java.sun.com/developer/technicalArticles/releases/j2se15/`

Bloch J (2008) Effective Java, 2nd edn. Prentice Hall, `http://java.sun.com/docs/books/effective/`

Brandt J, Guo PJ, Lewenstein J, Dontcheva M, Klemmer SR (2009) Two studies of opportunistic programming: interleaving web foraging, learning, and writing code. In: Proc. 27th Int'l Conf. Human factors in Computing Systems, pp 1589–1598

Cohen J (1960) A coefficient of agreement for nominal scales. Educational and Psychological Measurement 20(1):37–46

Cwalina K, Abrams B (2008) Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries (Second Edition). Addison-Wesley Professional

Dekel U, Herbsleb JD (2009) Improving API documentation usability with knowledge pushing. In: Proc. 31st Int'l Conf. Software Engineering, pp 320–330

Duala-Ekoko E, Robillard MP (2011) Using structure-based recommendations to facilitate discoverability in APIs. In: Proc. 25th European Conf. Object-oriented Programming, pp 79–104

Duala-Ekoko E, Robillard MP (2012) Asking and answering questions about unfamiliar apis: An exploratory study. In: Proceedings of the 34th IEEE/ACM International Conference on Software Engineering, pp 266–276

Fantechi A, Gnesi S, Lami G, Maccari A (2002) Application of Linguistic Techniques for Use Case Analysis. In: Proceedings of the IEEE International Joint Conference on Requirements Engineering, pp 157–164

Hariri N, Castro-Herrera C, Cleland-Huang J, Mobasher B (2014) Chapter 17: Recommendation systems in requirements discovery. In: Robillard MP, Maalej W, Walker RJ, Zimmermann T (eds) Recommendation Systems in Software Engineering, Springer

Henß S, Monperrus M, Mezini M (2012) Semi-automatically extracting FAQs to improve accessibility of software development knowledge. In: Proc. 34th Int'l Conf. Software Engineering, pp 793–803

Javadoc (2001) Javadoc. `http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html`

Jeong SY, Xie Y, Beaton J, Myers BA, Stylos J, Ehret R, Karstens J, Efeoglu A, Busse DK (2009) Improving documentation for esoa APIs through user studies. In: Proc. 2nd Int'l Symp. End-User Development, pp 86–105

Jurafsky D, Martin JH (2008) Speech and Language Processing, 2nd edn. Prentice Hall

Kawrykow D, Robillard MP (2009) Improving API usage through detection of redundant code. In: Proc. 24th Int'l Conf. Automated Software Engineering, pp 111–122

Kim J, Lee S, won Hwang S, Kim S (2009) Adding examples into Java documents. In: Proc. 2009 Int'l Conf. Automated Software Engineering, pp 540–544

Kof L (2007) Scenarios: Identifying missing objects and actions by means of computational linguistics. In: RE, pp 121–130

Landis JR, Koch GG (1977) The measurement of observer agreement for categorical data. Biometrics 33(1):159–174

Maalej W, Robillard MP (2013) Patterns of knowledge in API reference documentation. IEEE Transactions on Software Engineering 39(9):1264–1282

Manning CD, Raghavan P, Schütze H (2008) Introduction to Information Retrieval. Cambridge University Press

Monperrus M, Eichberg M, Tekes E, Mezini M (2011) What should developers be aware of? An empirical study on the directives of API documentation. Empirical Software Engineering Online

Neuendorf KA (2002) The Content Analysis Guidebook. Sage

Nykaza J, Messinger R, Boehme F, Norman CL, Mace M, Gordon M (2002) What programmers really want: results of a needs assessment for SDK documentation. In: Proc. 20th annual Int'l Conf. Computer Documentation, pp 133–141

Pagano D, Maalej W (2011) How do developers blog? An explorative study. In: Proc. 8th Working Conf. Mining Software Repositories, p 10

Pandita R, Xiao X, Zhong H, Xie T, Oney S, Paradkar A (2012) Inferring method specifications from natural language API descriptions. In: Proceedings of the 34th IEEE/ACM International Conference on Software Engineering, pp 815–825

Panichella S, Aponte J, Di Penta M, Marcus A, Canfora G (2012) Mining source code descriptions from developer communications. In: Proceedings of the 20th IEEE International Conference on Program Comprehension, pp 63–72

Robillard MP (2009) What makes APIs hard to learn? Answers from developers. IEEE Software 26(6):27–34

Robillard MP, DeLine R (2011) A field study of API learning obstacles. Empirical Software Engineering 16(6):703–732

Shepherd D, Fry ZP, Hill E, Pollock L, Vijay-Shanker K (2007) Using natural language program analysis to locate and understand action-oriented concerns. In: Proc. 6th Int'l Conf. Aspect-oriented Software Development, pp 212–224

Stylos J, Myers BA (2008) The implications of method placement on API learnability. In: Proceedings of the 16th ACM SIGSOFT International Symposium on the Foundations of Software Engineering, pp 105–112

Stylos J, Faulring A, Yang Z, Myers BA (2009) Improving API documentation using API usage information. In: Proc. 2009 IEEE Symp. Visual Languages and Human-Centric Computing, pp 119–126

Tan L, Yuan D, Krishna G, Zhou Y (2007) /*icomment: bugs or bad comments?*/. In: Proceedings of twenty-first ACM SIGOPS Symp. Operating Systems Principles, pp 145–158

Xiao X, Paradkar A, Thummalapenta S, Xie T (2012) Automated Extraction of Security Policies from Natural-Language Software Documents. In: Proceedings of the 20th ACM SIGSOFT International Symposium on the Foundations of Software Engineering

Ye Y, Fischer G (2002) Supporting reuse by delivering task-relevant and personalized information. In: Proceedings of the 24th ACM/IEEE International Conference on Software Engineering, pp 513–523

Ying AT, Robillard MP (2014) Developer profiles for recommendation systems. In: Robillard MP, Maalej W, Walker RJ, Zimmermann T (eds) Recommendation Systems in Software Engineering, Springer

Zhong H, Zhang L, Xie T, Mei H (2009) Inferring resource specifications from natural language API documentation. In: Proc. 2009 Int'l Conf. Automated Software Engineering, pp 307–318