CLASSIFYING AND RECOMMENDING KNOWLEDGE IN REFERENCE DOCUMENTATION TO IMPROVE API USABILITY

by

Yam Bahadur Chhetri

School of Computer Science
McGill University, Montreal

July 2012

A thesis submitted to McGill University
in partial fulfillment of the requirements of the degree of
Master of Science

Abstract

Reference documentation is an important source of information on API usage. Programmers, however, can easily overlook reference information because of its tedious nature, and because the information they seek can be buried among irrelevant or boiler-plate text. We propose to detect and recommend fragments of API documentation relevant and important to a task. We categorize pieces of information in API documentation based on the type of knowledge they contain. From the pieces that contain knowledge worthy of recommendation, we extract the composition and the pattern of words, and use the patterns to automatically find new pieces that contain similar knowledge. In an evaluation study, with a training set of manually-classified reference documentation of about 1 000 API elements, we could issue recommendations with about, on average, 90% precision and 69% recall.

Résumé

La documentation de référence est une source importante d'information sur l'usage d'une API. Cependant, les programmeurs peuvent négliger cette information que l'information recherchée se trouve enfouie au milieu de texte passe-partout et sans pertinence. Nous proposons de détecter et recommender les fragments de documentation d'API pertinents à une tâche donnée de façon automatique. Nous catégorisons les morceaux d'information dans la documentation d'API en fonction du type de savoir qu'ils renferment. À partir des morceaux de savoir digne de recommendation, nous extrayons des patrons de mots, puis utilisons ces patrons pour trouver automatiquement de nouveaux morceaux qui renferment un savoir similaire. Nous présentons les résultats d'une évaluation expérimentale de notre système effectuée à partir de plus de 1 000 morceaux d'API, où nous trouvons que notre système offre de recommendations adéquates 90% du temps avec un taux de rappel de 69%.

Acknowledgments

I thank my advisor, Martin Robillard, for his insights and guidance on every detail of this thesis. Martin's commitment to perfection was always a source of inspiration for me. I also thank my SWEVO lab mates — Bart, Annie, Peter, Gias, and Gayane — for all the productive discussions and feedback all along, and for making the lab a wonderful place to live and learn. I am also thankful to Mathieu Boespflug for the French abstract. Finally, for their continued support, I am always grateful to my parents.

Yam B. Chhetri
McGill University
July 2012

Contents

List of Figures

List of Tables

Chapter 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, such as the Java Standard Edition libraries or the .NET framework. Reusing APIs saves time and mitigates the risk of defects in implementing an equivalent feature from scratch. Using large APIs, however, is often challenging to many programmers [13], [28], [30], [34]. This challenge can be attributed to factors like interdependencies between multiple APIs, obscure API naming convention, low cohesion of an API, or lack of information on how to use them efficiently.

Providing extensive documentation for the APIs can help programmers understand the APIs better [28]. Documentation is thus an important constituent of APIs in particular and software projects in general. There exist different types of software documentation, such as reference documentation, code comments, tutorials, and white papers. Each of these types of documentation serves a specific purpose. For instance, API reference documentation, such as Javadocs,[1] provides information specific to individual API elements, whereas a tutorial, such as the Java Tutorial,[2] provides information used to accomplish an end-to-end task [2].

While extensive reference documentation can be useful for determining how to use

[1]By *Javadocs* we mean the documentation comment used in Java programs that describes the code and is extracted by the Javadoc tool. For example, http://docs.oracle.com/javase/6/docs/api/

[2]http://docs.oracle.com/javase/tutorial/

an API element, the content of the documentation can quickly become boilerplate. Research has shown that most programmers resort to reference documentation only after they are unsuccessful with information from many other sources [25]. Hence, by the time these programmers refer the reference documentation, large parts of the content has already become irrelevant because the programmers are equipped with a basic understanding of the use of the API element. For these programmers, the irrelevant information in the documentation includes those related to the functionality of the API element, the purpose and the basic properties of the element, the domain knowledge required to understand the element, etc. What these programmers need are fragments in the documentation that provide information not observable from experimenting with the API element [25], [24]. Such fragments are, however, spread throughout the documentation, and identifying them manually is tedious. In this thesis, we propose a technique to identify such documentation fragments automatically.

Using a grounded approach, we studied and identified properties of pieces of information present in API reference documentation and **classified them into three broad categories: those that are *must know*, those that are *good to know*, and those that *can be ignored*** by a programmer who has already identified the corresponding element as relevant to a task. We created a set of properties for characterizing the pieces of information that belong to the first two categories. We then created a coding guide and performed an empirical validation of the properties with an external participant, and observed high agreement on the classification of Javadocs of 300 API elements. **This classification of pieces of information in API reference documentation forms the first contribution of the thesis**.

In order to identify such pieces of information automatically, we devised a technique to identify the structure and the patterns of usage of words in them, because we often observed similarity in the usage of words in certain pieces of information that convey similar message. We extracted patterns of words for the *must know* and the *good to know* pieces of information from Javadocs of 900 random elements from Java SE 6 APIs, and automatically identified similar pieces on Javadocs of 120 random elements with a high precision; we employ natural language processing (NLP) and linguistic analysis techniques in the process. **This automated detection of**

important pieces of information in the reference documentation forms the second contribution.

In order to improve API usability in programming situations, we created 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. **This forms the third contribution**. We evaluated the approach end-to-end with sample Java code blocks extracted from ten open source systems. Our evaluation shows that with a training set consisting of about 1 000 manually-classified Javadocs, we can issue recommendations with about, on average, 90% precision and 69% recall. Furthermore, the recommended knowledge represents a small subset — about 14% — of the documentation relevant to a programming (recommendation) context. We further verified the recommendations on code blocks from a popular book that recommends efficient ways of programming in Java, 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.

1.1 Motivating Example

The following situation illustrates the need for classifying and recommending API documentation.

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

```
public class Run implements Runnable {
    public void run() {
      //some operation
    }
    public static void main(String args[]) {
      (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, as shown below, is not efficient in terms of resource management and performance.

```
...
(new Thread(new Run())).start();
(new Thread(new Run())).start();
(new Thread(new Run())).start();
...
```

To tackle some of these performance issues, Java Standard Edition 5.0 introduced[3] the `java.util.concurrent` package, which offers improved support for concurrency and, in particular, the Executor framework in `java.util.concurrent`. 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 {
   public void execute(Runnable r) {
       r.run();
   }
}
...
Executor executor = new AnExecutor();
executor.execute(new Run());
executor.execute(new Run());
executor.execute(new Run());
...
```

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

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

but 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 Javadocs of the three Java concurrency packages.

Our research provides an initial approach for surfacing this information and recommending it to programmers. In this situation our approach would leverage two key

[3]http://java.sun.com/developer/technicalArticles/releases/j2se15/

pieces of information: linguistic patterns to detect that the quoted sentence above is important, and the fact that the sentence contains the code word "`Thread`," which we resolve to a class initially used by the programmer.

1.2 Overview

In the remaining chapters, we describe the properties of the information categories, and present a technique to automatically detect pieces of information of the *must know* and the *good to know* types (Chapter 2). We introduce the Eclipse plug-in *Krec*, that is used to identify important pieces of information in the documentation of the API elements used in an input code block (Chapter 3). We present and discuss the evaluation results, and the threats to validity (Chapter 4), discuss related work (Chapter 5), and conclude the thesis (Chapter 6).

Chapter 2
Knowledge Items in Reference Documentation

2.1 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 SE 6 Javadocs to elicit the properties that can help us distinguish information we can recommend.

2.1.1 Concepts and Terminologies

Reference documentation is mostly composed of text. In this thesis 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,

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

> *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 specifically associated with 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 such a text segment as a **knowledge item** (KI).

2.1.2 Knowledge Items

Based on extensive observation and use of documentation, we determined that reference knowledge relevant to a task falls into one of two categories: *must know*, and *good to know*. We refer to the KIs in the *must know* category as **indispensable** KIs. These are the pieces of information which the programmers cannot afford to ignore, such as, the *caveats*, or the *threats*, in using certain API elements. We refer to the *good to know* category of KIs as **valuable** KIs. These KIs could highlight the benefit of using one method over another to achieve a similar objective. Figure 2.1 shows our model of documentation content.

Some common high-level properties of both these categories are that the pieces of information should be non-obvious and surprising for most programmers, and that they should have the potential to impact the decisions of the programmer. 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*

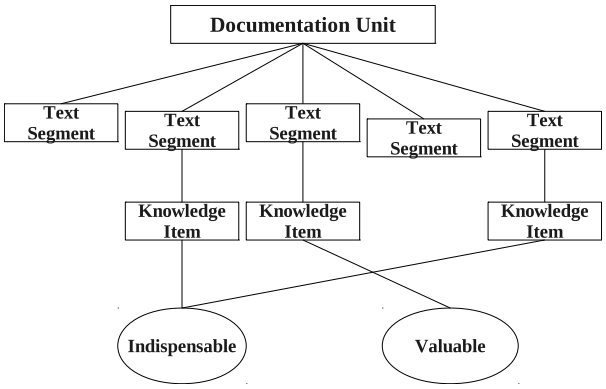

Figure 2.1: The documentation content model

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 [28], those of others in the field [22], [11], and some established principles [2]. We further expanded these properties using a grounded approach by closely studying the Javadocs of numerous API elements, followed by multiple iterations of refinement.

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 Javadocs. The characteristics of the two knowledge categories are described next.

Indispensable Knowledge Items

This is the type of KI that is essential to be aware of in order to use an API element. 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 subcategories below.

Usage Directives. It specifies 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.*

or

> *If you reimplement this method, you should also reimplement ...*

It is mandatory for the programmer to follow the specified guideline. Since we are interested in what the programmers *should* or *should not* do when using the API element, directives that are not mandatory constraints are not classified as *indispensable* in our work. An example of such a directive is "*may be null*," which essentially means that any value is permissible. An example of a mandatory constraint is "*must not be null*." Hence, for our case, the former is not an *indispensable* KI while the latter is.

Hard Constraints. It specifies hard constraints or specific requirements. For example,

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

The programmer cannot deviate from the specified requirement, or else the API element would be unusable or would throw a runtime error.

Threats. It specifies usage of certain protocols, which would otherwise 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.*

As a consequence of such threats, the programmer is obliged to follow the protocol or the specified convention.

Valuable Knowledge Items

This is the type of KI that conveys helpful and beneficial information. Programmers who ignore this category of information are likely to use the API sub-optimally [17], or spend an inordinate amount of time looking for information. As opposed to *indispensable* KIs, ignoring *valuable* KIs may not result in immediate defects but could have long-term repercussions. A *valuable* KI has properties related to one of the subcategories below.

Alternative API elements. It recommends 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`.

Such recommendations usually exist in the reference documentation of API elements that are developed in new releases and that have better features than a corresponding API element in an earlier release. Also, different API elements could be relevant to two different programming tasks with similar objectives.

Dependent API elements. It recommends 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 would help programmers figure out API elements that are dependent on the one the programmer is working with [12].

Improvement Options. It recommends actions that could lead to improvement in functionality or non-functional properties like 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. It recommends best practices, which would ideally 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

After we identify all the KIs from the reference documentation and classify them as either *indispensable* or *valuable*, the remaining content can be ignored, chiefly because they state facts that are unsurprising for programmers who have already selected the element, like the basic objective of an API element [10]. We do not claim that the information in those content has no value, but it ranks lower in terms of

the importance of the message it conveys. Assuming the programmer has already selected an API element and has a basic understanding of what it does [10], ignoring this category of information would neither have an immediate nor a long-term impact on the usage of the element.

Some of the properties of information in this category are:

Obviousness. A piece of information that is obvious from the name of the API element, for example, for the method `getAudioClip(URL url, String name)`, the following line in its Javadoc contains an obvious piece of information,

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

Unsurprising. A piece of information that is unsurprising for most programmers [10], for example, the summary sentence of API elements, that provides a high-level objective or functionality of the element [2].

Predictability. A piece of information that is predictable based on the context, for example,

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

2.1.3 Reliability Assessment

For the purpose of manually coding information, the description of the categories must be *reliable*. Reliability indicates with what consistency two independent coders (persons) would assign the same category to the same text segment [23].

We conducted a preliminary validation of the reliability of the coding guide on smaller test sets consisting of the Javadocs of 30 random API elements from the Java SE 6 APIs, over two separate rounds of coding. We then validated the properties in the coding guide with an external participant, a member of our lab not directly related to this research, with three rounds of validation on 77, 74, and 148 API elements, randomly selected from the Java SE 6 APIs and excluding those already included in the preliminary analysis. The non-uniformity in the number of API elements in each iteration was because the coding was counted at the end of a fixed period of one week without regards to the total count in the period. The reason for three iterations was to analyze the disagreements at the end of each iteration and fix common sources of

ambiguity in the coding guide, if any. The coding task involved the *manual* process of reading the assigned Javadocs, identifying text segments that satisfy the properties, and assigning them to the appropriate KI category.

We used the Cohen's Kappa metric (k) [8] to measure the reliability between two coders. Unlike a simple percent agreement calculation, Cohen's Kappa takes into account the chance agreement, i.e., the chance of two coders randomly selecting or rejecting a given text segment as a KI purely by chance. As per existing work, values of 0.61-0.80 for k are considered to indicate *substantial* agreement between two coders, and values of 0.81-1.00 are considered *almost perfect* [20].

The coding guide stated that a text segment is constituted by grouping sentences that provide a single unit of information. However, there were occasional disagreements between the coders in deciding what constituted a text segment. In order to resolve such a disagreement between two coders, we employed 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 one with the more number of sentences to form the 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.

The measures of the agreement between one of the authors and the external participant are presented in Table 2.1. The column *API* represents the total number of API elements coded in the corresponding iteration, and T is the total number of text segments across the Javadocs of all the API elements in the iteration. Text segments were counted using the two heuristics mentioned above for those selected by at least one of the coders, and manually examined by the authors for the rest. The number of instances of such fragment misalignment due to the first reason stated above were 5, 5, and 8, in the three iterations, and were 1, 0, and 4 due to the second reason. Thus, in total 23 misalignments had to be manually reconciled to compute agreement

Table 2.1: Reliability assessment

Iter.	*API*	*T*	**Ind.**	**Val.**	**Rest**	**Dis.**	*k*
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**	**0.80**

values, i.e., only 4% of the total text segments eventually identified.

We calculated k 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 2.1, 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. These are the cases where one of the coders selected a particular KI category for a text segment, and the other selected the other KI category or rejected it. We elicit the specific number of instances of such disagreements in Table 2.2. The values in the brackets represent the disagreement instances in the three iterations. For example, the value {1,2,2} in the **Valuable** row and the **Indisp.** column indicates that the number of instances when coder 1 selected the *valuable* KI category for a text segment and coder 2 selected the *indispensable* category are 1, 2, and 2 in the three iterations. The overall disagreement was 9.2%, i.e., there were disagreements for 53 text segments out of the total 578. The overall value of **0.80** for k, however, indicates substantial agreement between the two coders and we conclude that the properties for the KI categories are well-defined and that the instructions in the guide are clear.

Table 2.3 shows the agreement for the individual KI categories. In the individual category, the column t_s indicates the total text segments that are selected by both the coders, and the column t_d indicates the text segments selected by either of the two coders but not both. There is substantial agreement in both the categories.

Table 2.2: Instances of disagreement

	Indisp.	**Valuable**	**Rest**	**Total**
Indisp.	-	{0,0,2}	{2,3,3}	**{2,3,5}**
Valuable	{1,2,2}	-	{4,4,5}	**{5,6,7}**
Rest	{1,2,5}	{5,2,10}	-	**{6,4,15}**
Total	**{2,4,7}**	**{5,2,12}**	**{6,7,8}**	-

Table 2.3: Agreement on KI categories

Iteration	**Indispensable**			**Valuable**		
	t_s	t_d	k	t_s	t_d	k
1	22	4	**0.84**	50	9	**0.82**
2	28	8	**0.74**	42	8	**0.80**
3	38	12	**0.73**	90	12	**0.85**
Total	**88**	**24**	**0.77**	**182**	**29**	**0.82**

Table 2.4: Causes of major disagreements (%)

Reason	Iter. 1	Iter. 2	Iter. 3	Avg.
Best Practices	8	13	21	**14**
Property	38	33	26	**32**
Alternate API	15	13	16	**15**

Disagreement Analysis

At the end of the first two iterations, we noted if there were more than one disagreement due to a similar reason, for example, ambiguity in one specific property. Since a common property of any KI is that it should involve the programmer to make a decision regarding the usage of the API element, it was at times ambiguous to ascertain if certain text segments involved such an action. This was noticed in the following two types of information:

1. There were disagreements in understanding if certain **best practices** required a programmer decision. For example, for the following best practice, one coder assumed that it implicitly demanded a programmer decision and classified it as a KI, while the other did not.

   Updating an existing `LineBreakMeasurer` *is much faster than creating a new one.*

2. It was often uncertain if an API element's **property and purpose** involved a decision from the programmer. For example,

   Unlike sets, lists typically allow duplicate elements.

Surprisingly, we noticed that it was sometimes not obvious to detect **alternate API** recommendation. For example, one of the coders failed to include the following KI,

The methods inherited from the `BlockingQueue` *interface are precisely equivalent to* `BlockingDeque` *methods as indicated in the following table.*

The percentage of disagreements, in each iteration, due to these three reasons are presented in Table 2.4. These were, however, not significant enough to impact the overall agreement.

2.2 Automatic Detection of Knowledge Items

KIs are tedious to find manually, and there is a lot of documentation. The reference documentation for Java 6 SE alone comprises over 2.5 million words. We found it impractical to manually extract all KIs from reference documentation, so we are experimenting with techniques to find them automatically. The exact technical details in our approach are being continually improved. Here we report on the most recent stable iteration, which captures all the fundamental ideas we currently rely on.

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

The primary 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. Second, and more importantly, 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 code-like terms, or groups of words like *use* and *instead*. When we extracted such words from manually identified KIs and created additional combination of words by including those in the synonym set of each of the original words, we were able to generate multiple patterns, which provided us the means to find new KIs automatically; we used the WordNet[2] dictionary to find synonym sets.

The existence of common linguistic patterns in Javadocs is not surprising. Java programmers usually abide by the conventions specified in the Javadoc principles, which include *style guides*, *description formats*, or *the patterns in the use of terminologies* [2]. Some of these conventions state that the assertions in the specification need to be implementation independent, the descriptions should be descriptive and not prescriptive, and that the class or interface descriptions should ideally only state

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

the object and not the subject [2]. Such conventions make it possible to identify the words that are representative of the information conveyed in a sentence. Hence, from the manually identified KIs, we extracted the words that are important to the knowledge conveyed in the sentence, 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,*

which provides an efficient alternative to unpack a `Pack200` instance. For this KI, if we extract the words *may*, *efficient*, and the code-like term `File`, then one of the text segments, from a different Javadoc, that would match this pattern is the following,

> *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`. This text segment too can be classified as a *valuable* KI; it provides an alternate recommendation that is potentially better than the one the programmer is working with, to achieve a similar objective.

In our technique, a *KI pattern* is simply a set of words, optionally including a special word that is a placeholder for code terms, such as `File`. A sentence will match a KI pattern if it contains all the words in a pattern (or a synonym).

The remainder of this chapter is divided into three parts. First, we discuss the approach to find KIs in training sets of Javadocs. Second, we present the technique to extract patterns of words from the KIs. Third, we describe the process of using the patterns for automatic detection of new KIs in test sets consisting of large numbers of Javadocs.

2.2.1 Knowledge Item Identification

Patterns are derived from valid KIs. We developed some automated support to facilitate the identification of valid KIs from which we eventually derive patterns. Our tool support for KI identification consists of two components: the *preprocessor* and the *automated filter*. These are shown on the left side of Figure 2.2. In our case, the

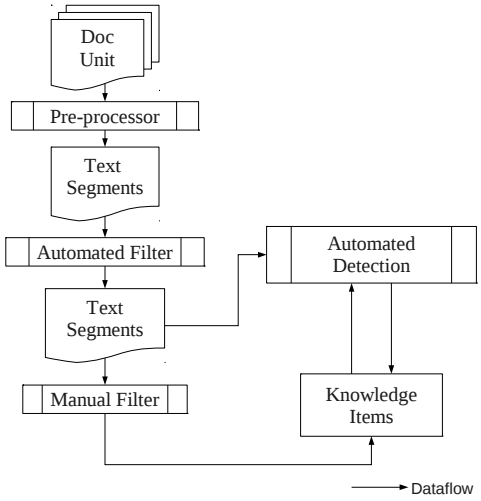

Figure 2.2: The knowledge item identification process

documentation units in the figure are represented by Javadocs.

Pre-processor

The *pre-processor* performs the following operations:

- segregates the Javadocs belonging to individual API elements, i.e., it separates the Javadocs of methods and fields from the Javadoc of the *type* containing them.
- strips the HTML tags from the Javadocs of individual API elements and extracts the plain text, and identifies possible code-like terms, from the HTML code tags or based on additional heuristics for those not within code tags. The heuristics include identifying camel-cased tokens, identifying tokens that are equivalent to the name of the API element the Javadoc belongs to, etc.
- converts the raw text into a list of sentences. We used the `PunktSentenceTokenizer` module in the Natural Language Toolkit[3] (NLTK) libraries to identify sentences in a text.

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

- eliminates non-alphanumeric characters like curly braces at the beginning or at the end of a sentence.
- groups sentences that begin with a *conjunction*, e.g., *thus*, *hence*, *therefore*, etc., with their preceding sentence because they would represent continuation of the same information.

Thus, the pre-processor outputs a list of text segments.

Automated Filtering

We use an *automated filter* to eliminate some of the text segments output by the pre-processor. The elimination is based on several heuristics as described below; we created a pipeline of filters that performs the elimination.

- In Javadocs, the first sentence of "each member, class, interface, or package description" contains a high-level summary of what the element is supposed to do [2]. 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.
- Javadocs 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 eliminate some of the independent clauses mentioned at the bottom of the Javadocs of most API elements, especially methods. These include the clauses *Throws*, *See Also*, *Since*, *Specified by*, and the first sentence in *Returns*. See Appendix A for the rationale behind eliminating each of these clauses.

The output of the automated filter is a cleaned up and simplified version of the documentation text that can then be manually inspected and classified.

Manual Filtering

The automated filters described above eliminate text segments that are unlikely to satisfy any of the properties of the two KI categories. This is, however, only a part of the total text segments that need to be eliminated. The rest of the elimination is done manually, i.e., the text segments that get past the automated filters are looked at manually and compared against the properties defined for the two categories. The text segments that satisfy the properties are then classified into the appropriate category. Details of the manual filtering process followed to create a training set for automatic detection of KIs is described in Chapter 2.2.3.

The output of the knowledge item identification phase is a list of KIs, as shown in Figure 2.2. Also depicted in the figure is the use of such KIs to automatically detect KIs from other documentation units. For this, however, the automated detector needs patterns of words in the manually detected KIs, in order to match similar KIs. The process of extracting such patterns is described next.

2.2.2 Identifying Patterns in Knowledge Items

Manually-classified KIs can be collected into a *training set* that is then used to identify patterns. 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 steps required to transform a KI into a corresponding pattern are described in the following subsection and are shown in Figure 2.3, in the same order as they occur.

POS Tagging

In this phase, the sentences that are part of the KIs in the training set are tagged with their part of speech (POS) using a *tagger*; we used the NLTK implementation

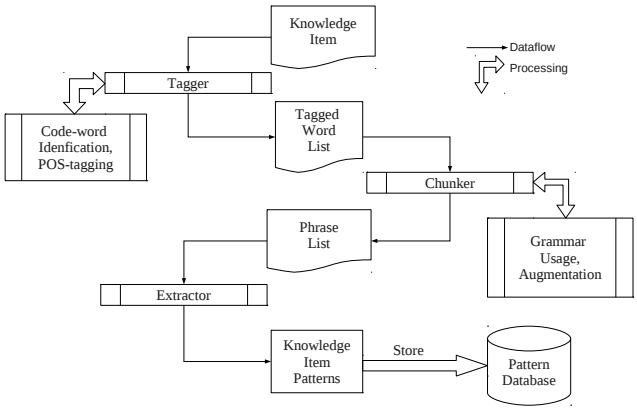

Figure 2.3: The pattern identification process

of the `Penn Treebank` tagger to tag words with their POS. We further augmented the tagging technique to handle code-like terms and tag them distinctly. The output of this phase is a list of KIs consisting of words each tagged with their POS.

To evaluate the performance of our POS tagger, we took the first 50 Javadocs that were part of the reliability assessment exercise, and separately created two tag lists: one using the Penn Treebank tagger, and the other a gold standard using a different tagger followed by manual evaluation, and compared the two. To create the gold standard, we used the default NLTK tagger based on the Brown corpus to produce an initial tag list. The Penn Treebank and the Brown corpus differ in important ways [36]. Unlike the taggers based on Penn Treebank, those based on the Brown corpus do not take into consideration the syntactic context of a word in a sentence [36].

For the 50 Javadcos, we generated about 1 500 tagged tokens. For those generated using the default tagger, we manually inspected the output, and corrected false tags, if any. On comparing the final tagged tokens with those generated by the Penn Treebank, we observed an accuracy of 91%, where the accuracy was calculated by comparing each tagged token. This is lower than the accuracy that the tagger could have achieved on a regular english text, which in general is above 97% for most taggers, because in Javadocs, especially those of methods, the first sentence usually starts with a verb, which is uncommon syntax. For example, in the sentence "*creates*

an empty border that takes up no space," our tagger assumed "*creates*" to be a *noun (plural)* whereas it is a *verb (present)*. Since we disregard the first sentence from consideration as explained earlier (Automated Filtering), such false tags pose less risk in our objective of identifying phrases (Chunking) from text segments.

Chunking

The tagged KIs are then passed to a *chunker* that identifies different phrases present in the sentences composing the KIs. These phrases could be either of *noun*, *verb*, *adjective*, or *modal* phrase. In each of these phrases, there would be a headword accompanied by other supporting words. For example, a noun phrase could have a noun accompanied by a determiner and an adjective. The usefulness of such phrases is that it makes it possible to extract just the headword leaving out the supporting words in the phrase.

Code-like terms, such as `addSource` in the KI "*a correct way to write the* `addSource` *function is to clone the source*," are treated as nouns. A modal phrase has 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*.

The chunker relies on grammatical rules to create the different phrases. Our initial set of rules include the standard English rules for the different phrase types. For example, $\{NP{:}\{<DT|PP\$>?<JJ|VBN><NN|CW><VBN>*\}\}$[4] is a noun phrase. It indicates that a phrase that has zero or one occurrence of a determiner or a possessive pronoun, followed by an adjective or a past participle verb, and then a noun or a code-like term is a noun phrase. Hence, we can extract the noun or the code-like term from this phrase and eliminate other words. The elimination process is a combination of automated and manual steps; the automated rules remove the obvious supporting terms like the articles, and we manually remove the words that are classified as headwords by the chunker but are still less significant.

[4]DT: determiner, PP$: possessive pronoun, JJ: adjective, VBN: verb, past participle, NN: Noun, CW: Codeword

To determine the initial grammatical rules for automated chunking, we used the CoNLL 2000 corpus,[5] which is annotated with POS and chunk tags, and used the NLTK corpus module to access them. With observations from several KIs in the training set, we augmented the rules to handle code-like terms and 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 phrase extracts the words `JComponent` and *subclasses*, and a rule for verb phrase, the word *override*. Our additional rules further identifies `JComponent` as a code-like term from its POS tag, and our modal phrase extracts the word *must*.

Extraction

From each phrase detected in a KI, all supporting words (non-headwords) are automatically eliminated, i.e., not included in the pattern. The remaining list of headwords is then produced as the output of the automated process. The final step requires a human to look at the list of headwords and eliminate any word that does not usefully capture the essense of the KI.

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*).

2.2.3 Automated Detection of Knowledge Items

To put the idea of recommending API documentation in practice, we must solve the problem of finding all KIs for a given API, e.g., the Java 2 SE class libraries,[6] which consists of 206 packages, 3 869 types, 28 724 methods, and 6 158 fields. The Javadocs of all these API elements total 2 632 232 words in 194 204 sentences.

[5] http://www.cnts.ua.ac.be/conll2000/
[6] http://docs.oracle.com/javase/6/docs/api/

Our basic strategy for meeting this challenge is to manually find *validated KIs* for a small random *training set* of API elements (with the tool support described above), generate patterns from them, and then use these patterns to find the KIs in all remaining unseen Javadocs. A key aspect of this strategy, however, is that we 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 allows us to measure 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 the 299 KIs produced as part of the development of the classification scheme (Chapter 2.1).

2. Generate a *test set* by randomly sampling 20 unseen Java elements.

3. Apply all known patterns to this test set. Calculate the precision and recall. Precision is calculated by manually determining how many instances were falsely identified as KIs. Recall is calculated by reading the entire Javadocs of all items in the test set, and identifying false negatives (i.e., missing KIs).

4. 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 Java elements. Manually identify KIs in them, and add the corresponding patterns to the pattern database.

6. Go to step 2.

The high-level process is shown in Figure 2.4. The sample input in the figure consists of both the training and the test sets. The training sets are passed through the semi-automated approach of knowledge item and pattern identification processes, and the test sets through the automated detector that is provided with the patterns. The vetting step is where the test sets are further scrutinized for missing KIs.

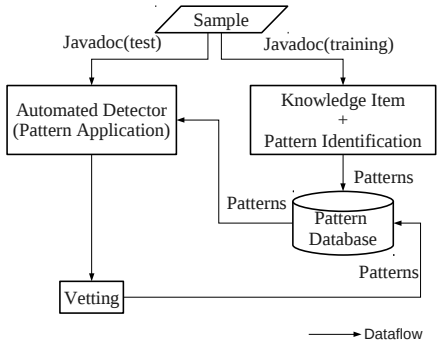

Figure 2.4: The iterative pattern application process

Results

Table 2.5 shows the results of using the automated detector on six test iterations with 20 Javadocs in each test. The *matched KI* column indicates the total number of sentences identified as KIs in the test Javadocs. Of these potential KIs, the number of true positives is indicated in column *correct matches*. *Precision* shows the ratio. The *New KIs* column lists the false negatives and *Recall* summarizes the ratio. After six test iterations, we had evaluated 799 API elements in the training sets and 120 in the test sets. For the total of these 919 API elements, which is about 2.4% of the total API elements in Java SE 6 libraries we collected 556 *unique KI*, and 361 unique patterns; 142 *indispensable* and 219 *valuable*. We find this to be a relatively low yield for patterns, which explains that recall values remain modest. This result is explained by the relatively small size of our test sets, which is constrained by the high cost of manual validation. When applied to a larger corpus, patterns yield many more instances (details below). More importantly, the precision of **92%** for the 43 automatically matched KI is relatively high. A high precision-low recall tradeoff is desirable in our case because programmers currently do not have any documentation recommendations. We surmise that a few correct recommendations will improve over nothing at all, while numerous bogus recommendations have a potential to be aggravating. 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,

Table 2.5: Iterative testing results

Test Round	Matched KI	Correct Matches	New KIs	Precision (%)	Recall (%)
1	3	3	7	**100**	**30**
2	4	4	8	**100**	**33**
3	5	4	2	**80**	**67**
4	17	14	9	**82**	**61**
5	10	9	11	**90**	**45**
6	4	4	5	**100**	**44**
Total	**43**	**38**	**42**	**92**	**47**

we opted to apply the pattern database at this point in the process.

General Application

From the remaining 97.6% of the unseen Javadocs, 56% of them (21 700) had one or more sentences in their Javadocs after the automated filtering step. We then applied the 361 patterns to automatically identify potential KIs in their Javadocs. The automated detector was able to detect one or more KIs in the Javadocs of 8 396 (38.7%) of them. We stored the automatically detected KIs in a final KI database.

Table 2.6 presents the details of the KIs, generated by the automated detector and those manually vetted. Out of the 8 396 elements for which there is one or more KI in the corpus, 75% are methods, 20% are types, and the rest fields. Out of all the automatically generated KIs, 80% belongs to Javadocs of methods.

The most effective pattern was the combination of the words *CW*, *must*, and *pass*, where *CW* is a placeholder for a code-like term; it matched 1072 KIs across all the Javadocs, which is 7.6% of the total KIs automatically identified. 265 out of the total 361 patterns matched one or more KIs. The average instances of a match per pattern was 52 and the standard deviation 116.

Table 2.6: Knowledge item corpus

	Total	**With**	**Manual**		**Auto**	
		KI	**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

Chapter 3
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 from the Javadocs of the API elements in the code. *Krec* uses the KI corpus generated by the automated detector, presented in the earlier chapter, to identify KIs. *Krec* also recommends KIs from the Javadocs of 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`. Since code terms are tagged appropriately in the corpus, a search for a code-like term only matches those with a code tag and not a regular word.

Figure 3.1 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.

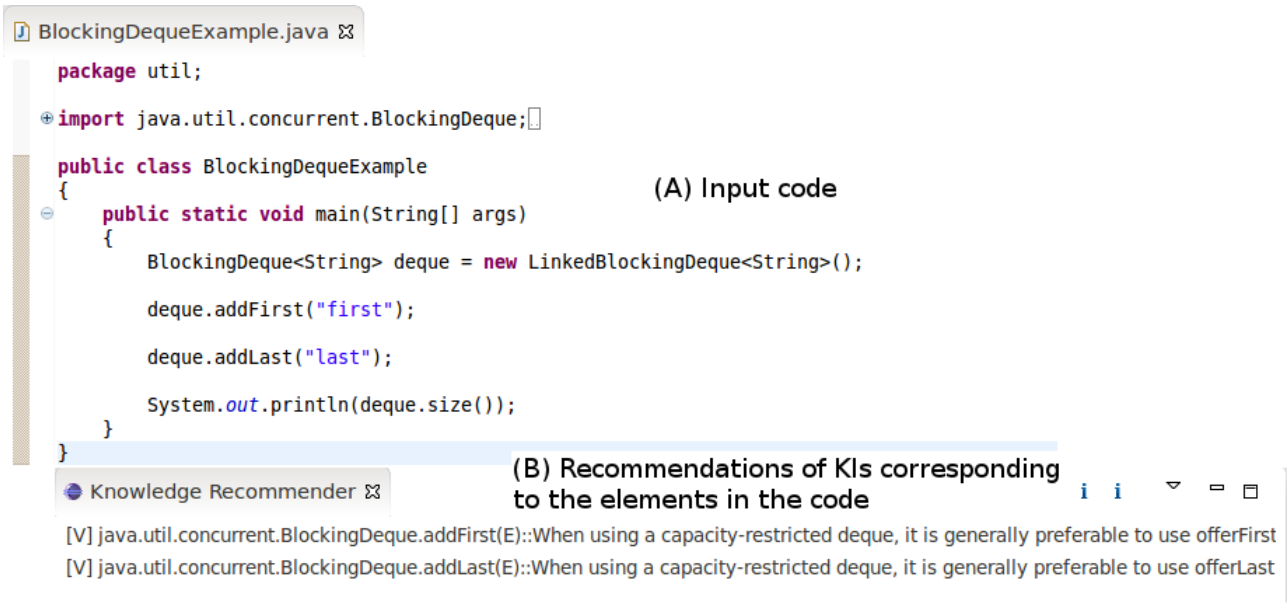

Figure 3.1: Krec in action

Chapter 4
Evaluation

We evaluated the impact and the use of KIs recommended by *Krec* in varied programming situations. For input code blocks, we analyzed the recommendations from *Krec* using the following metrics:

(a) **Precision**. This measures the precision of the automated detector by analyzing the correctness of the generated KIs. For all the generated KIs, we also flag appropriately those identified manually as part of the initial training phase (Chapter 2.2.3), and those identified by the automated detector. We measured separately the precision of the KIs from the Javadocs of elements in the code, and those that belong to elements not in the code but mention the elements.

(b) **Recall**. This measures the number of missed KIs. It is obtained by manually studying the details of the Javadocs for API elements; we measured the recall for only those elements that were present in the code blocks.

(c) **Extraneous Information**. This measures the number of text segments that the programmer who has already selected an API element can afford to ignore. These are the text segments in the documentation that are not KIs.

We further evaluated the recommendations by targeting code examples with one or more *inherent flaw*; the majority of the flaws in the snippets are not obvious to

most programmers. The fix for the flaws come as recommendations from third-party sources, and we compared those recommendations with the KIs recommended by *Krec*. For such code snippets, we assess the following metric:

(d) **Match**. This measures whether or not the recommended KIs contain the expected recommendations; an *expected recommendation* for a sample of code block is third-party recommendation on the means to improve the code or API usage in the sample. For each input code sample, this metric had one of the following values:

Yes If the expected recommendation matches a KI recommended by *Krec*.

No If the expected recommendation is present in the documentation but *Krec* does not recommend it.

NA If the expected recommendation is not present in the documentation.

4.1 Design

To evaluate the usefulness of KI recommendations in different programming situations, we collected **Java code samples** from production settings, and suitable **targeted code examples** from a popular book on Java programming.

Production Code

These are representative of code in use by open source programmers. Since our corpus contains KIs present in the Javadocs of API elements in the Java Development Kit (JDK) libraries, we picked systems that use the JDK APIs in different ways. We chose ten open source systems from varying domains as our target (Table 4.1). We used the following characteristics to choose the systems:
Domain. The systems represent different domains and each serves a different purpose. The version of each of the systems used in the evaluation and their brief purposes are presented in Table 4.1.

Table 4.1: 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 objects between Java and XML

Size. They vary significantly in their size, the details of which are extracted from Ohloh.[1] For instance, JUnit contains around 26K lines of code, compared to Tomcat, which contains over 1.6 million lines of code. The average lines of code across the ten systems is about 815K with a standard deviation of about 798K.

Use of JDK APIs. These systems differ in their use of the JDK APIs, for example, Joda-Time uses only the fundamental JDK APIs like the *containers*, whereas JEdit relies on the Abstract Window Toolkit libraries, and XStream heavily uses the *Reflection* APIs.

Since the evaluation required extensive manual investigation, we used stratified sampling to create a representative code sample from each system to input to *Krec* for evaluation. To form the strata, from each of the ten systems, we selected twenty method definitions, each consisting of more than five lines of code and using at least one JDK API element. In addition, the method definitions are as widespread in the system as possible, i.e., if the system is composed of twenty or more packages, we picked the methods from classes representing twenty *different* packages. Table 4.2 depicts the distribution of API elements in the sample code. The column *Total*

[1] http://www.ohloh.net/

Table 4.2: Distribution of API elements in the samples

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

LoC indicates the total lines of code across the twenty method definitions in each system, and *Total APIs* indicates the sum of the occurrence of all the *relevant* JDK API elements; we obtained the *relevant* elements by filtering out pervasive elements, i.e., those that are equivalent to *stopwords* in natural language text, for example, `PrintStream`, `System`, etc. This list is in Appendix B. **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 are only present in the sampled method definitions in each system, i.e., elements in the **Unique APIs** column for a system are only present in the sampled method definitions of that system. Hence, on average 40%, i.e., 50.6 elements, are unique to each system in the sample.

We input the method definitions to *Krec* and measured the recommendations against the proposed metrics.

Targeted Examples

The limitation with the production code is that we could not measure the *Match* metric because we did not have third-party oracle of recommendations to compare our recommendations against. Hence we looked for code samples that recommended means to make API usage more efficient. We found such code samples, i.e., code with one or more inherent flaws or with inefficient API usage, and the recommended fix or improvement in API usage, in the Effective Java book (EJ) [1].

EJ contains several complete code samples, each containing a defect, the fix for which is to obey an associated rule. Each of these rules signify a programming rule or a best practice. Specifically, EJ contains 78 rules intended to make the most effective use of the Java programming language [5]. We broadly categorized these 78 rules into two types:

1. those that recommend **programming best practices**, e.g., "*never do anything time-critical in a finalizer*" [5, p. 27].

2. those that recommend means of **effective usage of the fundamental Java**

Table 4.3: Evaluation results - production code

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

SE libraries, e.g., "*always override* `toString()`" [5, 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 [1], and input it to *Krec*.

4.2 Results

Production Code

Table 4.3 shows the evaluation results on the method definitions selected from the ten open source systems. The table shows the average precision and recall across

elements in all the method definitions in the individual systems. For the first three metrics, we only discuss the results from the evaluation of the open source systems, because the EJ items, with small code sample, may not be representative of results for *precision*, *recall*, and *extraneous information*, in general.

Precision. For the ten open source systems, on average, *Krec* was able to recommend a KI with **90% precision per system**. On average, it identified 1 *indispensable* KI for every 5.2 API elements and 10.2 lines of code, and 1 *valuable* KI for every 2.1 elements and 4.2 lines of code. The precision is consistently above 85% across all the systems, except JDT, with 83.8%.

The results in Table 4.3 illustrate the average performance of our approach, and aggregate multiple occurences of the same recommendations. 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 4.3 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 4.4). Across the twenty method definitions in the ten systems, there was a total usage of 1 277 API elements. Out of the total elements, 660 were distinct, i.e., after removing multiple occurrences of elements across methods and systems. From these 660 elements, 31 were part of the training set, and *Krec* was able to automatically generate one or more KIs in 186 of the remaining elements. In these 186 elements, *Krec* automatically identified 215 KIs, out of which 163 were correct matches, which gave a **micro-averaged precision**, i.e., precision after summing up values from all the elements, of **75.8% per element**. The values from the individual elements were computed by looking into their respective Javadocs, and analyzing each text segment for the presence of KI. This is lower than the average precision per system reported in Table 4.3 because Table 4.3 contains manually tagged KIs and multiple occurrences of some KIs. The macro-averaged precision, i.e., the average of

Table 4.4: Auto generated KIs per element

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

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 Javadocs, the number of KIs ranges from 1 to a maximum of 8, with more than 50% of the cases having 1 KI. Since precision (and recall) suffer from high variance (e.g., 0 or 100 on Javadocs with 1 KI) on such small samples, we report only the micro-averaged value.

Krec did not recommend any KI from Javadocs of elements not in the code, either because the code samples were not representative of such cases or there was no matching pattern.

Recall. We evaluated this metric by manually looking into the Javadocs of the API elements in the input code, and figuring out the missing KIs, i.e., those that are not present in the KI corpus. *Krec* achieved a **recall of 68.6%** of the KIs across the 20 method definitions in the ten systems. On average, it missed 1 *indispensable* KI for every 2.7 that it found, and 1 *valuable* KI for every 2 that it found.

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 Javadocs, and were not part of the random training set.

For the 660 distinct elements across all the systems, *Krec* missed 164 KIs; the **micro-averaged recall** on a per element basis is **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.

Extraneous Information. To measure this metric, we evaluated the total number of text segments in the Javadocs of all the *relevant* API elements in the input code sample, and noted the ones that are not KIs, i.e., those that we ignored. This is not necessarily indicative of useless information but of those that have lower significance to the programmer, who has already selected an element, in terms of the message it conveys. In Table 4.3, the column *Total TS* indicates the sum of all the text segments across the Javadocs of all the *relevant* JDK API elements in the code. We noticed that, on average, 13.6% of the documentation contains all the KIs, which implies that 86.4% of the documentation contain low impact information. This finding is in fact consistent with the Pareto Principle of 80/20, which in software engineering roughly states that 80% of the engineering deals with 20% of the requirements; in software testing, it is believed that 80% of bugs are caused by 20% of code.

Targeted Examples

Table 4.5 shows the evaluation results on the sample code associated with the EJ rules; the Java files are downloaded[2] from the EJ web location [1]. For the reason stated earlier, we used the code associated with the EJ items to evaluate the *match* metric only.

Match. We used this metric to measure the match of an expected recommendation for a code block with a KI generated by *Krec*. Out of the 78 items in EJ, we found 11 to represent API usage rules. 8 of these 11 have the expected recommendations in the documentation of one or more API elements in the associated code. Out of these 8, *Krec* was able to correctly match the recommendations for 6 of them; the 6 cases represented constraints, best practices, and alternate API recommendation.

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

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

Table 4.5: Evaluation results - targeted examples

EJ Items	KI Rec.		KI Missed		Expected
	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

1. Item 29, where it talks about the use of the method `asSubClass` in `Class` in order to safely cast an object [5, p. 146]. The equivalent fact in the Javadoc of `asSubClass` is present as a *purpose*, which did not match any of the existing KI patterns, because the patterns do not represent purposes and properties of API elements.

2. The sample code in Item 36, which uses an overloaded `equals` instead of `overridding` it [5, p. 177], hence *Krec* could not identify the constraint associated with the original `equals` method.

In conclusion, for the match metric, we claim that *Krec* can consistently match the expected recommendation on a code block, that uses APIs inefficiently, with a KI from the documentation, if the expected recommendations exist in the form of threats, alternative API recommendation, or best practices.

4.3 Discussion

Our work is the first of a kind that uses a semi-supervised learning methodology to automatically recommend fragments of API documentation.

Types of Information in API Documentation

A primary outcome of the work is the reliable classification of the pieces of information in API documentation. Since the categories of information contained in a documentation unit is subjective and could differ with context and perspective, we ensured, aided by our experience, and an elaborate analysis of varying documentation units, that we capture all the essential categories. Though at a high level, the *indispensable* and the *valuable* categories seem generic, the properties identified in each of the categories are specific and broad, and ensures that a coder can in fact reliably identify a text segment satisfying a property, and that we do not miss any property that is important for consideration.

Our coding guide underwent multiple iterations of refinement until we observed little or no change in the agreement level between two coders. The agreement analysis (Chapter 2.3.1) showed that an external coder can reliably agree on the properties and identify the appropriate text segments.

Another outcome of the work is the identification of text segments in a paragraph or a documentation unit. In our reliability assessment of around 300 Javadocs, we observed only 4% disagreement between two coders in deciding what constitutes a text segment. For the automated identification of text segments, we relied on conjunctions to group sentences together. This, however, in exceptional cases, can still fail to group sentences that present a single unit of information, and could render some text segments unintelligible if observed in isolation. A remedy to this would be a careful display of the recommended text segments to the programmer. For example, instead of recommending a text segment in isolation, we could highlight the text segment within the documentation unit, so that if the programmer feels that the text segment does not convey a concrete meaning, the programmer can refer to the sentences preceding the text segment to understand the information conveyed. Using

additional heuristics to automatically identify text segments remains future work.

Existence of Patterns in API Documentation

The existence of patterns of word usage in reference documentation is dependent on the API developers obeying certain documentation convention [2]. We showed that pattern identification is a reliable and inexpensive way to capture different types of information. We used a semi-automated approach to identify patterns of words in a text segment. Though the automated chunking technique removed obvious supporting words in a text segment, thereby reducing the number of words to less than 50% in majority of the text segments, we relied on a manual intervention to get the final pattern of words.

It would also be interesting to capture negative patterns, i.e., if we identify a text segment that says it is mandatory for the programmer to use a certain type of variable, it would also be interesting to capture a pattern that requires the programmer not to use certain type of variable.

A limitation with the chunking technique that we used is in automatically identifying patterns that only make sense when two or more words are put together. For example, if a text segment contains the phrase "one of the following," it would usually imply a mandatory action from the programmer to have to use one of the following values, but the chunker can not group these words into a single phrase, and we end up missing an important pattern. Improvement in this chunker property remains an interesting future work.

4.4 Threats to Validity

The code samples from EJ do not represent code in production settings, hence the match of the expected recommendation cannot be generalized. In order to measure the usefulness of the recommendations, the samples needed to have both a defect and third party solutions. Another approach to evaluate the usefulness of the KIs in real programming situations would be a user study with sampled programming problems.

The process of identifying KIs is semi-automated and involves manual intervention at two critical phases: identifying the *indispensable* and the *valuable* KIs in the training set, and then identifying the essential headwords from the list output by the automated filter. Though we observed high agreement with an external participant on identifying the KIs, the agreement could vary when evaluated with more participants and those with varying background. Also, identifying the essential headwords could be subjective; a conservative selection might increase false positives, while selecting more headwords as part of a KI pattern would reduce recall.

The coding guide is based on the assumption that the documentation follows the style specified in the Javadoc principles [2]. However, if an API developer deviates from the convention, the properties of the KIs specified in the guide may not yield the desired results.

Although we evaluated the automated detection of KIs on varying domains, KIs could still be unique to certain domains, which would minimize the recall from patterns extracted from other domains. For example, Joda-Time uses `Date`, `Calendar`, and `Locale` APIs, that are specific to the domain, hence it did not have matching patterns in the training set. Such domains that use unique APIs could threaten the validity of our approach.

Chapter 5
Related Work

This thesis builds on studies of API usability, API documentation, and applications of natural language processing in software engineering.

5.1 API Usability

Researchers have proposed metrics to measure the usability aspects of APIs. One set of metrics are simplicity, productivity, error-prevention capabilities, and consistency in the design and the functionality [33]. Proper design is an important aspect of API usability because APIs are meant for reuse, and a poorly designed API will leave spoils in all the dependent code [4]. Hence, researchers have proposed a few design choices to help improve API usability [4], [34], [30]. Stylos and Clark investigated the use of parameters in object constructors and found that programmers were more efficient when the constructors did not require parameters [30]. Ellis et al. studied the use of Factory design pattern in APIs and found that programmers preferred constructors over static factory methods [13]. Stylos and Myers evaluated the implications of method placement choices in API design, and found that if a class from which users generally start to explore an API had methods that referenced other classes in the API, it significantly enhanced the productivity of the programmer [34]. Stylos et al. further demonstrated the impact of design choices on API usability in a case

study of user-centered API redesign in an industrial context [32]. The above studies focus on improving API usability by recommending better design decisions, but such decisions are not always feasible for existing APIs, because once an API becomes public, its developers cannot change them at will, without affecting dependent code. As opposed to design decisions, our work focuses on making existing APIs more usable by extracting useful information and suppressing irrelevant ones from their reference documentation.

Researchers have also developed various plug-ins to improve code completion features in popular IDEs (Integrated Development Environments) to improve API usability [7], [12]. Bruch et al. used example code from repositories to rank the elements accessible to an object in the IDE [7]. Duala-Ekoko and Robillard proposed *API Explorer* that uses structural relationship between API elements to recommend relevant methods on other objects referenced indirectly [12]. Though our work has a similar objective, that of making APIs more usable, we employ a different approach to it, that of recommending specific information from the reference documentation. We can also recommend dependent API elements, like *API Explorer* does, if such information is present in the reference documentation of either the main type [12], the helper type [12], or other closely related API elements.

5.2 API Documentation

The impact of documentation on the usability of APIs has also been an area of active research [31], [6], [27], [11], [28]. Researchers have made proposals to make API documentation easily accessible and understandable to programmers.

Though the reference documentation is an important form of API documentation, studies have found that programmers use reference documentation only when they fail to get enough information from other possible sources [25]. 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 [25], and Jeong et al. identified the importance of explaining starting

points to increase the quality of the documentation [16]. 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 worked on highlighting directives present in Javadocs of several major APIs [11]. 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 have to include such tags in the documentation. Also, it would be difficult to identify directives in existing documentation that are void of such tags. In contrast, we automatically identify possible directives as well as other important forms of information, for example, alternative API recommendation, in the reference documentation, by training a detector of such information.

Monperrus et. al. performed an extensive empirical study and identified all possible directives in three large Java projects, viz., JDK, JFace and Commons Collections [22]. 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.

Stylos et al. proposed *Jadeite*, which studies source code and statistically provides recommendations to the programmers on the most used classes, constructors, methods, and objects [31]. 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. proposed *eXoaDocs*, which uses code snippets, mined from search engines, to improve documentation, by integrating the code with the text [18]. 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.

5.3 NLP in Software Engineering

Some of the research in software engineering have used Natural Language Processing (NLP) to infer contracts or specifications from documents [37], [3], [35]. Kof used computational linguistics to identify missing objects and actions in requirements documents [19]. Likewise, Fantechi et al. used a linguistic approach to analyze functional requirements expressed by means of textual use cases [14]. While these approaches focused on requirements documents, another line of research used source code or documentation to infer important information using linguistic approaches. Shepherd et al. used NLP techniques to locate and comprehend concerns in source code [29]. Arnout and Meyer proposed a technique to infer invariants, like preconditions and postconditions, from the documentation [3]. Tan et al. proposed *iComment*, that extracts specifications from comments in source files [35]. Zhong et al. proposed *Doc2Spec* that infers resource specifications from API documentation [37]. Our approach uses API reference documentation for linguistic analysis but does not focus on inferring specifications. We only rely on the inherent structure of API reference documentation to identify important types of information.

Recent work has also investigated how the natural-language text found in on-line blogs and forums can be analyzed to support software engineering activities. Examples include the work of Pagano and Maalej [26] and that of Henß et al. [15]. Pagano and Maalej 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. [15] 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).

Chapter 6
Conclusion and Future Work

To identify important pieces of information in API reference documentation, we presented an approach to classify the pieces in the documentation based on the type of knowledge items they contain. We proposed two types of knowledge items: the *indispensable* (*must know*) and the *valuable* (*good to know*).

In a training set of Javadocs of around 1 000 API elements from the Java SE 6 libraries, we identified pieces of information that satisfy a property of either of these two categories of knowledge items, and we extracted word usage patterns. We applied these patterns to automatically detect similar pieces of information in the Javadocs of the remaining API elements in the Java SE 6 libraries.

We evaluated the approach using an Eclipse plug-in, *Krec*, on code blocks of method definitions extracted from 10 varied open source systems. With the training set of 1 000 manually-classified Javadocs, we could issue recommendations with about, on average, 90% precision and 69% recall. Furthermore, the recommended knowledge represents a small subset — about 14% — of the documentation relevant to a programming (recommendation) context. We verified that obvious recommendation from textbook examples could be recommended, with a success rate of 6 out of 8 cases. Our approach involves many steps, which we plan to continue to evaluate and improve. However, our initial results provided us with the evidence that word usage pattern is a cost-effective approach for finding *indispensable* and *valuable* knowledge items in reference documentation.

Other areas for future work are improvements in the display of the knowledge items to the end user, ideally in a manner that is consistent with the existing IDE features like the *content assist* in Eclipse. A user study with sampled programming problems could be used to evaluate the usefulness of the recommendations in a new programming task. Though it is a relatively harder problem, identifying and recommending context-sensitive knowledge items is an interesting area to explore.

Appendix A
Automated Filter of Clauses

The automated filtering described in Chapter 2.2.1 eliminates the following clauses from the Javadocs of API elements:

- **Throws**. The purpose of the *throws* (or equivalently *exception*) tag in Javadocs is to indicate which exceptions the programmer must catch or might want to catch [2]. As a principle of Javadocs, checked exceptions must be included in a *throws* clause of the method so that the compiler can know which exceptions to check. For the programmer though, such exceptions are evident in the method signature. Even otherwise, exceptions are named such that it is intuitive to determine the type and the cause, e.g., `FileNotFoundException` indicates an error in trying to access a non-existent file. For these reasons, we eliminate the *throws* clause from considering that it would contain *indispensable* or *valuable* KIs.

- **See Also**. The *see also* clause only lists other API elements without any explanation, hence it can neither contain *indispensable* nor *valuable* KIs.

- **Since**. Likewise the *since* clause only mentions the project version.

- **Specified by**. Likewise the *specified by* clause only mentions the name of the interface that originally contains the method.

- **Returns**. The first sentence in the *returns* clause can be eliminated because it often contains redundant information about the element that is returned [2]. However, subsequent text segments, if any, could contain important information, hence we take them into consideration.

Appendix B
Pervasive API Elements

The following is the list of pervasive API elements omitted from consideration from code blocks in the evaluation:

`java.lang.String`,

`java.lang.System`,

`java.lang.Object`,

`java.lang.Package`,

`java.lang.Void`,

`java.lang.Error`,

`java.lang.Deprecated`,

`java.lang.Override`,

`java.lang.SuppressWarnings`,

`java.lang.System.out`,

`java.io.PrintStream*`.

Appendix C
Coding Guide

You will be given a list of Javadocs of API elements in the Java SE 6 libraries. An API element could be a *type* (class, abstract class, interface, enum, or annotation type), a field, or a method. Your task is to read the given Javadocs, and look for the presence of two types of knowledge items (KIs), *indispensable* and *valuable*, which are described below. For each of the Javadocs that you think contains either an *indispensable* or a *valuable* KI, you need to identify the sentence or the groups of sentences that contain them.

In the context of our project, a *knowledge item* is a piece of information in the documentation that is of significant *value*. E.g., for the method `RemoteObject.getRef()`, the line in its Javadoc that says "*returns the remote reference for the remote object*" contains information that is obvious from the method name, hence it is of little value and does not represent a KI. The line following it that says "*... an instance of* `RemoteRef` *should not be serialized outside of its* `RemoteObject` *wrapper instance or the result may be unportable*," however, contains information of value and we classify it as representing a KI.

The two categories of KIs that we are interested in are those that contain a piece of information that the user *must know*, which we term *indispensable*, and those that contain a piece of information that is *good to know*, which we term *valuable*. One commonality between the two categories is that the piece of information should *not* be evident from the name and the signature of the API element.

Indispensable Knowledge Items

This is the type of information that **cannot be ignored**; users who ignore this category of information would either encounter compilation or runtime errors or are most likely to introduce bugs. *Indispensable* KIs instruct users to **perform mandatory action(s)**, explicitly or otherwise, to achieve basic functionality.

*Please note that we are looking for KIs that demand **actions from a programmer** as opposed to those that mention the purpose or the functionality of an API element or an overview of certain concept without requiring an action from the programmer. Beware that actions may not always be explicit and could be **implied within the KI**. E.g., a KI that says "it is not safe to call method X from method Y" implicitly instructs the programmer that the programmer should not call method X from method Y.*

We assign the following properties to *indispensable* KIs in Javadocs:

1. It specifies **mandatory** API **usage directives**. E.g.,
   - *By convention, the returned object should be obtained by calling* `super.clone`.
   - *This method should only be called by a thread that is the owner of this object's monitor.*
   - *The provider must supply an implementation class containing the following method signatures . . .*
2. It imposes **hard constraints** or **specific requirements**. E.g.,
   - *must not be null.*
   - *A valid port value is between 0 and 65535.*
   - *At most one field or property in a class can be annotated with* `@XmlAnyAttribute`.
   - *The string must match exactly an identifier used to declare an enum constant in this type.*
   - *The usage is subject to the following constraints . . .*

- *If an application needs to read a password or other secure data, it should use* `readPassword()` *or* `readPassword (String, Object...)`
- *The maximum number of arguments is limited by the maximum dimension of a Java array as defined by the Java Virtual Machine Specification.*
- *A character may be part of a Java identifier if and only if any of the following are true . . .*

3. It indicates ***potential*** **threats** or **warnings** if certain protocols are not followed, thereby (implicitly) instructing the programmer to follow them. *Information regarding thread-safety or synchronization issues usually satisfy this property.* E.g.,

   - *invoking methods . . . as well as the read, format and write operations on the objects returned by . . . may block in multithreaded scenarios.*
   - *A* `CannotProceedException` *instance is not synchronized against concurrent multithreaded access. Multiple threads trying to access and modify* `CannotProceedException` *should lock the object.*
   - *Note that this implementation is not synchronized. If multiple threads access an* `ArrayList` *instance concurrently, and at least one of the threads modifies the list structurally, it must be synchronized externally.*
   - *Warning: Serialized objects of this class will not be compatible with future Swing releases.*
   - `DataInputStream` *is not necessarily safe for multithreaded access.*

Please note that the following types of information **do not satisfy** the properties for *indispensable* KIs:

- Directives that are not necessarily constraints, e.g., "*can be null*" or "*may be null*" for a parameter value, which essentially means that the parameter can have any possible value.

- Constraints in the values returned from a function as opposed to constraints while setting certain parameter values, e.g., the sentence "*a valid port value is between 0 and 65535*" puts a constraint on the user to have to use one of the values between 0 and 65535, but the sentence "*returns a value between 0 and 65535*" provides a piece of information without imposing any constraint on the user, hence the latter is not classified as a KI in our study.

Valuable Knowledge Items

This is the type of KI that conveys **good to know** information. Users who ignore this category of information are likely to **use the API sub-optimally**, or spend an inordinate amount of time looking for information. Unlike *indispensable* KIs, ignoring *valuable* KIs will not result in immediate bugs, but are nevertheless *almost* equally important to be considered in order to use the API optimally and prevent potential future issues.

Please note that we are looking for KIs that demand ***actions from a programmer*** *as opposed to those that mention the purpose or functionality of an API element or an overview of certain concept without requiring an action from the programmer. Beware that actions may not always be explicit and could be* ***implied within the knowledge****. E.g., a KI that says "it is preferable to use method X over method Y," implicitly recommends the programmer to use method X over method Y.*

We assign the following properties to *valuable* KIs in Javadocs:

1. It **recommends alternate** API elements to accomplish the same objective (often efficiently). *This property is also valid for cases where certain API elements are deprecated and the documentation suggests not using them and recommends alternate API elements instead.* Note that pieces of information that **compare API elements** also satisfy this property because comparisons usually give the programmers options to choose from two or more API elements. E.g.,

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

- *In general, `String.toLowerCase()` should be used to map characters to lowercase.*
- *The correct way to write `PutField` data is by calling the `ObjectOutputStream.writeFields()` method.*
- *If a new `Character` instance is not required, this method should generally be used in preference to the constructor `Character(char)`, as this method is likely to yield significantly better space and time performance by caching frequently requested values.*
- *The constant factor is low compared to that for the `LinkedList` implementation.*

2. It **suggests API elements** or a **series of API elements** that could be used to perform a task. *Please note that this is only true if the suggested API element(s) is **different** from the one the Javadoc belongs to, otherwise it would just be functionality or purpose of the concerned API element.* E.g.,

   - *To specify the use of a different class loader, either set it via the `Thread.setContextClassLoader()` api or use the `newInstance` method.*
   - *Use `getName` to get the logical name of the font.* [This information is present in `getFamily`, i.e., different from `getName`, hence it becomes *valuable.*]
   - *To support all Unicode characters, including supplementary characters, use the `canDisplay(int)` method or `can- DisplayUpTo` methods.*
   - *An instance of `Current` can be obtained by the application by issuing the `CORBA::ORB::resolve_initial_references("POACurrent")` operation.*
   - *Instances of this class are generally created using a `SSLServerSocketFactory`.*

3. It indicates actions that could lead to **improvements** in non-functional properties, like performance, scalability, maintainability, etc. E.g.,

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

4. It recommends **best practices** that should be adhered to. *Note that this is different from the hard constraint property of indispensable KI. Hard constraints are mandatory and will result in immediate programming failure, whereas best practices are optional, but are always best followed.* E.g.,
   - *This means that you are advised not to use this class and, in fact, it may not even be available depending on your* `JAXB` *provider.*
   - *Validator has been made optional and deprecated in* `JAXB` *2.0. Please refer to the javadoc for* `Validator` *for more detail.*
   - *Thus in general applications are strongly discouraged from accessing methods defined on* `SAXSource`*.*
   - *This method is intended to be used only by event targeting subsystems, such as client-defined* `KeyboardFocusManagers`*. It is not for general client use.*
   - *Clients generally should call* `isTransformed()` *first, and only call this method if* `isTransformed` *returns* `true`*.*
   - *The class supplied by the provider does not have to be assignable to* `javax.xml.bind.JAXBContext`*, it simply has to provide a class that implements the* `createContext` *APIs.*

An example of a piece of information that describes a concept without requiring an action from the programmer, hence not a *valuable* KI in our study, is "*different fields will be set depending on the type of validation that was being performed.*"

Additional Information

The following types of information should be ignored, i.e., they can neither be *indispensable* nor *valuable* KIs:

1. **Information about functionality or purpose**: Please note that the obvious high-level functionality or purpose of an API element can neither be considered *indispensable* nor *valuable*. This is because we are interested in information that is **surprising** and **not obvious** from the name and the signature of the API element. In addition, we wish to find pieces of information that are interesting in the case where the programmers have already chosen the API element that they need. Hence we assume the programmer already has a basic understanding of what the API element does, i.e., its high-level functionality or purpose. Therefore, the first line of documentation in the Javadoc of API elements, which by design contains a high-level summary of the functionality of the API element, can be ignored. Any line that mentions just the purpose of an API element should also be ignored. E.g., "*the* `ObjectOutputStream` *continues to be recommended for interprocess communication and general purpose serialization*" or "*code can use this to serialize or deserialize classes in a purposefully malfeasant manner.*"

2. **Non-concrete information in abstract classes and interfaces**: The documentation of abstract classes and interfaces often contain information that is applicable to only some of their implementations. E.g., a specific property of `List` could be applicable to `ArrayList` and not to `TreeList`. In such cases, only include information that is applicable to either all the implementations of the abstract class or interface or to a specific implementation. E.g., do not include sentences like "*some implementations prohibit null elements, and some have restrictions on the types of their elements*," which does not say for sure which such implementations are. In place of "some implementations," if there were phrases like "this implementation" or "all implementations," then include them.

3. **Redundant information in the same documentation unit**: If you find that the same KI is repeated within the documentation of an API element, please include only the first occurrence. In Javadocs, this usually occurs when a piece of information is mentioned in the description of an API element and repeated afterwards in the **Returns** clause.

4. **Generic external reference:** Avoid generic external reference that is common to multiple API elements. E.g., "*As of 1.4, support for long term storage of all JavaBeans has been added to the* `java.beans` *package. Please see* `XMLEncoder`*.*"

5. **If in doubt, leave out**: If you are not able to decide whether a piece of information contains a KI, leave it out.

Bibliography

[1] Effective Java second edition. `http://java.sun.com/docs/books/effective/`.

[2] Javadoc. `http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html`.

[3] Karine Arnout and Bertrand Meyer. Uncovering hidden contracts: The .NET example. *Computer*, 36(11):48–55, November 2003.

[4] Joshua Bloch. How to design a good API and why it matters. In *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*, pages 506–507, 2006.

[5] Joshua Bloch. *Effective Java (2nd Edition) (The Java Series)*. Prentice Hall PTR, 2008.

[6] Joel Brandt, Philip J. Guo, Joel Lewenstein, Mira Dontcheva, and Scott R. Klemmer. Two studies of opportunistic programming: interleaving web foraging, learning, and writing code. In *Proceedings of the 27th International Conference on Human factors in Computing Systems*, pages 1589–1598, 2009.

[7] Marcel Bruch, Martin Monperrus, and Mira Mezini. Learning from examples to improve code completion systems. In *Proceedings of the the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, pages 213–222, 2009.

[8] Jacob Cohen. A coefficient of agreement for nominal scales. *Educational and Psychological Measurement*, 20(1):37–46, 1960.

[9] Rylan Cottrell, Robert J. Walker, and Jörg Denzinger. Semi-automating small-scale source code reuse via structural correspondence. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 214–225, 2008.

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

[11] Uri Dekel and James D. Herbsleb. Improving API documentation usability with knowledge pushing. In *Proceedings of the 31st International Conference on Software Engineering*, pages 320–330, 2009.

[12] Ekwa Duala-Ekoko and Martin P. Robillard. Using structure-based recommendations to facilitate discoverability in APIs. In *Proceedings of the 25th European Conference on Object-oriented programming*, pages 79–104, 2011.

[13] Brian Ellis, Jeffrey Stylos, and Brad Myers. The factory pattern in API design: A usability evaluation. In *Proceedings of the 29th International Conference on Software Engineering*, pages 302–312, 2007.

[14] Alessandro Fantechi, Stefania Gnesi, G. Lami, and A. Maccari. Application of linguistic techniques for use case analysis. In *Proceedings of the 10th Anniversary IEEE Joint International Conference on Requirements Engineering*, pages 157–164, 2002.

[15] Stefan Henß, Martin Monperrus, and Mira Mezini. Semi-automatically extracting FAQs to improve accessibility of software development knowledge. In *Proceedings of the 2012 International Conference on Software Engineering*, pages 793–803, 2012.

[16] Sae Young Jeong, Yingyu Xie, Jack Beaton, Brad A. Myers, Jeff Stylos, Ralf Ehret, Jan Karstens, Arkin Efeoglu, and Daniela K. Busse. Improving documentation for esoa APIs through user studies. In *Proceedings of the 2nd International Symposium on End-User Development*, pages 86–105, 2009.

[17] David Kawrykow and Martin P. Robillard. Improving API usage through detection of redundant code. In *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering*, pages 111–122, 2009.

[18] Jinhan Kim, Sanghoon Lee, Seung won Hwang, and Sunghun Kim. Adding examples into Java documents. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 540–544, 2009.

[19] Leonid Kof. Scenarios: Identifying missing objects and actions by means of computational linguistics. In *RE*, pages 121–130, 2007.

[20] J. Richard Landis and Gary G. Koch. The measurement of observer agreement for categorical data. *Biometrics*, 33(1):159–174, March 1977.

[21] David C. Littman, Jeannine Pinto, Stanley Letovsky, and Elliot Soloway. Mental models and software maintenance. In *Papers presented at the first workshop on Empirical Studies of Programmers*, pages 80–98, 1986.

[22] Martin Monperrus, Michael Eichberg, Elif Tekes, and Mira Mezini. What should developers be aware of? An empirical study on the directives of API documentation. *Empirical Software Engineering*, Online Edition, 2011.

[23] K. A. Neuendorf. *The Content Analysis Guidebook*. Sage Publications, 2002.

[24] David G. Novick and Karen Ward. Why don't people read the manual? In *Proceedings of the 24th annual ACM International Conference on Design of Communication*, pages 11–18, 2006.

[25] Janet Nykaza, Rhonda Messinger, Fran Boehme, Cherie L. Norman, Matthew Mace, and Manuel Gordon. What programmers really want: results of a needs

assessment for SDK documentation. In *Proceedings of the 20th annual International Conference on Computer Documentation*, pages 133–141, 2002.

[26] Dennis Pagano and Walid Maalej. How do developers blog? An exploratory study. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, pages 123–132, 2011.

[27] Martin P. Robillard. What makes APIs hard to learn? Answers from developers. *IEEE Software*, 26(6):27–34, November 2009.

[28] Martin P. Robillard and Robert Deline. A field study of API learning obstacles. *Empirical Software Engineering*, 16(6):703–732, December 2011.

[29] David Shepherd, Zachary P. Fry, Emily Hill, Lori Pollock, and K. Vijay-Shanker. Using natural language program analysis to locate and understand action-oriented concerns. In *Proceedings of the 6th International Conference on Aspect-oriented Software Development*, pages 212–224, 2007.

[30] Jeffrey Stylos and Steven Clarke. Usability implications of requiring parameters in objects' constructors. In *Proceedings of the 29th International Conference on Software Engineering*, pages 529–539, 2007.

[31] Jeffrey Stylos, Andrew Faulring, Zizhuang Yang, and Brad A. Myers. Improving API documentation using API usage information. In *Proceedings of the 2009 IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 119–126, 2009.

[32] Jeffrey Stylos, Benjamin Graf, Daniela K. Busse, Carsten Ziegler, Ralf Ehret, and Jan Karstens. A case study of API redesign for improved usability. In *Proceedings of the 2008 IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 189–192, 2008.

[33] Jeffrey Stylos and Brad Myers. Mapping the space of API design decisions. In *VL/HCC*, 2007.

[34] Jeffrey Stylos and Brad A. Myers. The implications of method placement on API learnability. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 105–112, 2008.

[35] Lin Tan, Ding Yuan, Gopal Krishna, and Yuanyuan Zhou. /*iComment: bugs or bad comments?*/. In *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles*, pages 145–158, 2007.

[36] Ann Taylor, Mitchell Marcus, and Beatrice Santorini. The Penn Treebank: An overview, 2003.

[37] Hao Zhong, Lu Zhang, Tao Xie, and Hong Mei. Inferring resource specifications from natural language API documentation. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 307–318, 2009.