An Empirical Study on Evolution of API Documentation

Lin Shi[1], Hao Zhong[1], Tao Xie[3], and Mingshu Li[1,2]

[1] Laboratory for Internet Software Technologies, Institute of Software, Chinese Academy of Sciences, Beijing 100190, China
[2] Key Laboratory for Computer Science, Chinese Academy of Sciences, Beijing 100190, China
[3] Department of Computer Science, North Carolina State University, USA
{shilin,zhonghao}@itechs.iscas.ac.cn, xie@csc.ncsu.edu, mingshu@iscas.ac.cn

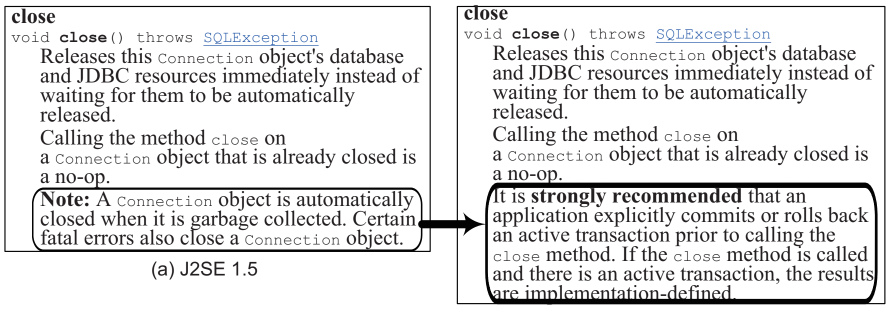
Abstract. With the evolution of an API library, its documentation also evolves. The evolution of API documentation is common knowledge for programmers and library developers, but not in a quantitative form. Without such quantitative knowledge, programmers may neglect important revisions of API documentation, and library developers may not effectively improve API documentation based on its revision histories. There is a strong need to conduct a quantitative study on API documentation evolution. However, as API documentation is large in size and revisions can be complicated, it is quite challenging to conduct such a study. In this paper, we present an analysis methodology to analyze the evolution of API documentation. Based on the methodology, we conduct a quantitative study on API documentation evolution of five widely used real-world libraries. The results reveal various valuable findings, and these findings allow programmers and library developers to better understand API documentation evolution.

1 Introduction

In modern software industries, it is a common practice to use Application Programming Interface (API) libraries (*e.g.*, J2SE[1]) to assist development, and API documentation is typically shipped with these API libraries. With API documentation, library developers provide documents on functionalities and usages of API elements (*i.e.*, classes, methods, and fields of API libraries), and programmers of library API client code (referred to as programmers for short in this paper) follow these documents to use API elements.

Due to various factors such as adding new functionalities and improving API usability, both API libraries and their documentation evolve across versions. For example, the document of the `java.sql.connection.close()` method in J2SE 1.5[2] has a notice that connections can be automatically closed without calling the `close` method (Figure 1a). In J2SE 1.6[3], library developers delete the notice, and emphasize the importance of calling the `close` method explicitly (Figure 1b). In practice, the preceding document of J2SE 1.5 is misleading, and causes many related defects. For example, a known defect[4] of the Chukwa project is related to unclosed `JDBC` connections. Existing research [16] shows that programmers are often unwilling to read API documentation

[1] `http://www.oracle.com/technetwork/java/javase/overview`
[2] `http://java.sun.com/j2se/1.5.0/docs/api/`
[3] `http://java.sun.com/javase/6/docs/api/`
[4] `http://issues.apache.org/jira/browse/CHUKWA-9`


close

`void close() throws SQLException`

Releases this `Connection` object's database and JDBC resources immediately instead of waiting for them to be automatically released.

Calling the method `close` on a `Connection` object that is already closed is a no-op.

Note: A `Connection` object is automatically closed when it is garbage collected. Certain fatal errors also close a `Connection` object.

(a) J2SE 1.5

close

`void close() throws SQLException`

Releases this `Connection` object's database and JDBC resources immediately instead of waiting for them to be automatically released.

Calling the method `close` on a `Connection` object that is already closed is a no-op.

It is **strongly recommended** that an application explicitly commits or rolls back an active transaction prior to calling the `close` method. If the `close` method is called and there is an active transaction, the results are implementation-defined.

(b) J2SE 1.6

Fig. 1. An example of API documentation evolution

carefully. If programmers miss the revision in J2SE 1.6, they may follow the old document in J2SE 1.5, and still introduce defects that are related to unclosed connections even after the document is modified. From the revision, library developers can also learn a lesson, since their API documentation contains misleading documents. In summary, revisions in API documentation are important for both client code development and library development.

As API documentation contains documents for hundreds or even thousands of API elements, revisions in API documentation could also be quite large in size. Due to the pressure of software development, it is difficult for programmers and library developers to analyze these revisions systematically and comprehensively. A quantitative study on evolution of API documentation can help programmers and library developers better understand the evolution, so there is a strong need of such a quantitative study. However, to the best of our knowledge, no previous work presents such a study, since textual revisions across versions are typically quite large, and analyzing these revisions requires a large amount of human effort and a carefully designed analysis methodology.

In this paper, we present an analysis methodology to analyze API documentation evolution quantitatively. Based on the methodology, we conduct a quantitative study on documentation provided by five widely used real-world libraries.

The main contributions of this paper are as follows:

– We highlight the importance of API documentation evolution, and propose a methodology to analyze the evolution quantitatively.
– Based on our methodology, we provide the first quantitative analysis on API documentation evolution. The results show various aspects of API documentation evolution. The results allow programmers and library developers to better understand API documentation evolution quantitatively.

The rest of this paper is organized as follows. Section 2 presents our analysis methodology. Section 3 presents our empirical results. Section 4 discusses issues and future work. Section 5 introduces related work. Section 6 concludes.

2 Analysis Methodology

To conduct a systematic and quantitative study on API documentation, we present a methodology to analyze documentation evolution. Given API documentation of two versions of an API library, we classify their revisions into various categories.

Table 1. Versions of selected libraries

Library	v1	v2	v3	v4	v5
J2SE	1.2.2	1.3.1	1.4.2	1.5	1.6
ActiveMQ	5.0.0	5.1.0	5.2.0	5.3.0	5.3.1
lucene	2.9.0	2.9.1	2.9.2	3.0.0	3.0.1
log4j	1.2.12	1.2.13	1.2.14	1.2.15	1.2.16
struct	2.0.14	2.1.2	2.1.6	2.1.8	2.1.8.1

Step 1: Identifying revisions. In this step, we first compare API documents of the two versions to find their revisions. In API documentation, we refer to a collection of words that describe an API element as a document. During evolution, an API element may be added, removed, and modified (see Section 3.3 for our results). As we focus on API documentation, we consider an API element modified when its declaration changes or its document changes. For a changed document, we refer to a pair of two associated sentences with differences as a revision. We divide a revision into one of the three categories: (1) an addition: a newly added sentence; (2) a deletion: a deleted sentence; (3) a modification: a modified sentence. Finally, we extract various characteristics such as word appearances and change locations from identified revisions.

Step 2: Classifying revisions based on heuristic rules. In this step, we first analyze a few hundred randomly selected sample revisions, and extract characteristics for various categories manually. For each category, we define heuristic rules according to these characteristics. For example, we find that different types of annotations (*e.g.*, `@see` and `@version`) are associated with different key words with specific fonts, so we define a corresponding heuristic rule to classify revisions by their annotations. With these heuristic rules, we classify revisions into various clusters by our heuristic rules. Revisions in the same cluster share similar characteristics.

Step 3: Refining and analyzing classified revisions. In this step, we tune our classifiers iteratively. In each iteration, we analyze inappropriately classified results to refine existing heuristic rules or to implement new rules for better classification. When classified results are accurate enough, we further fix remaining inappropriately classified results, and analyze results for insights on the evolution of API documentation (see Sections 3.1 and 3.2 for our results).

3 Empirical Study

In our empirical study, we focus on three research questions as follows:

RQ1: Which parts of API documentation are frequently revised?

RQ2: To what degree do such revisions indicate behavioral differences?

RQ3: How frequently are API elements and their documentation changed?

We use five widely used real-world libraries as subjects. The five libraries have 9,506,580 words of API documentation in total. For each library, we analyze the latest five stable versions as shown in Table 1. For J2SE, we do not choose some end-of-life versions (*i.e.*, J2SE 1.4.1, J2SE 1.4.0, and J2SE 1.3.0)[5]. We still choose J2SE 1.2.2 (an end-of-life version), since existing stable versions of J2SE do not have five releases. For ActiveMQ, we analyze its core API elements only. More details can be found on

[5] `http://java.sun.com/products/archive/j2se-eol.html`

our project site: `https://sites.google.com/site/asergrp/projects/apidocevolution`.

3.1 RQ1: Which parts of API documentation are frequently revised?

In this section, we present proportions of all types and some examples of these types. In total, we compared 2,131 revisions that cover the `java.util` package of J2SE, and all the other four libraries. We identify three primary categories of API documentation evolution, and for each category, we further identify detailed revision types, as shown in Figure 2. The vertical axis shows the three primary categories of revisions, and the horizontal axis shows the proportion for each category over the 2,131 revisions.

Finding 1: 45.99% of revisions are about annotations as follows.

In the official guidance of Java documentation[6], tags and annotations are different. By using tags, library developers can add structures and contents to the documentation (*e.g.,* `@since`), whereas annotations are used to affect the way API elements are treated by tools and libraries (*e.g.,* `@Entity`). In this paper, we do not distinguish the two definitions and use *annotation* to represent both of them for simplicity.

Version 19.25% A version consists of numbers and dates that indicate when an API element is created or changed, and a version is marked by the `@version` annotation or the `@since` annotation. With either annotation, the version of an API element is updated automatically when API code changes. As the two annotations cause to automatically modify documents, they cause a large proportion of revisions. It is still an open question on whether such version numbers are useful, since we notice that library developers systematically deleted the two annotations in the documentation of lucene 3.0.0.

Exception 8.21% Exception handling plays an important role in the Java language, and the exceptions of an API method can be marked by the `@throws` annotation or the `@exception` annotation. Revisions on exceptions often indicate behavioral differences, and Finding 4 in Section 3.2 presents more such examples.

Reference 7.60% The document of one API element may refer programmers to other API elements since these API elements are related. The reference relations can be marked by the `@see` annotation or the `@link` annotation. We find that at least two factors drive library developers to modify reference relations. One factor is that the names of referred API elements are modified. The other factor is that it is difficult even for library developers to decide reference relations among API elements. Besides other API elements, one document may even refer to some Internet documents using URLs. Library developers may also modify URLs across versions. For example, in the `java.util.Locale.getISO3Language()` method of J2SE 1.5, the URL of ISO 639-2 language codes is updated from one[7] to another[8].

Using the annotations such as `@see` or `@link`, one document may refer to other code elements or documents, and these code elements or documents may get updated

[6] `http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html`
[7] `ftp://dkuug.dk/i18n/iso-639-2.txt`
[8] `http://www.loc.gov/standards/iso639-2/englangn.html`

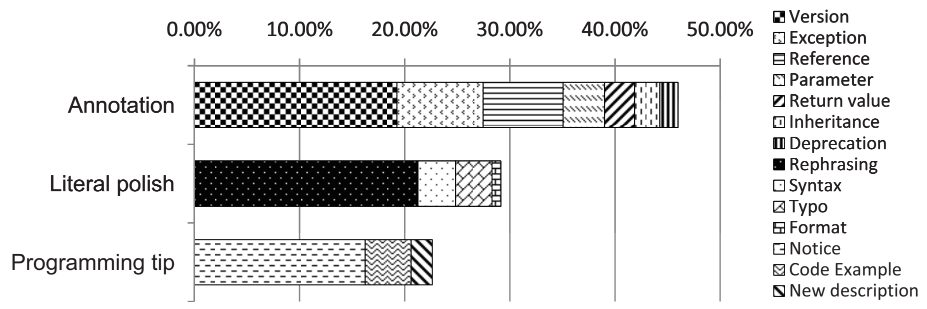


Fig. 2. Categories of revisions.

across versions. Code refactoring [15] is a hot research topic, but most of existing refactoring approaches address the problem of refactoring code partially, and its impacts on documents are less exploited.

Parameter 3.94% For each API method, the document of its parameters can be marked by the `@param` annotation, and library developers may add parameter documents to describe parameters. For example, in J2SE 1.3.1, library developers add a document to the `comp` parameter of the `java.util.Collections.max(Collection, Comparator)` method: "`comp` the comparator with which to determine the maximum element. A `null` value indicates that...".

Return value 2.86% The return value of an API method can be marked by the `@return` annotation, and library developers may add return-value documents to better describe return values. For example, in J2SE 1.4.2, library developers add a document to the `java.util.ArrayList.contains(Object)` method: "Return: `true` if the specified element is present; `false` otherwise".

Inheritance 2.39% Inheritances among classes and interfaces cause similar documents across them, and also their methods and fields. The document of a class often needs to be modified, when the document of its superclass is modified. Although library developers can use the `@inheritDoc` annotation to deal with similar documents caused by inheritance relations, we notice that only a small proportion of such similar documents are marked by the annotation.

Deprecation 1.74% An API element may become deprecated, and its document can be marked by the `@deprecated` annotation. When the document of an API element is marked as deprecated, some IDEs such as the Eclipse IDE explicitly warn that programmers should be careful to use the API element. When an API element becomes deprecated, library developers sometimes may suggest programmers to use alternative API elements. Although a deprecated API element is assumed to be deleted in later versions, we find that some deprecated API elements can become undeprecated again. For example, we find that eight API elements (*e.g.*, the `org.apache.lucene.search.function.CustomScoreQuery.customExplain()` method) get updated from deprecated to undeprecated in lucene 3.0.0.

Implication 1: Library developers take much effort to write annotations, and some annotations (*e.g.*, `@see`) are difficult to maintain. Some tools may be beneficial if they can help library developers write and update these annotations. Researchers may borrow ideas from existing research on code refactoring when implementing these tools.

Finding 2: 29.14% of revisions are literal polishes as follows.

Rephrasing 21.26% We find that library developers often rephrase documents to improve their accuracies. For example, in J2SE 1.2.2, the document of the `java.util.GregorianCalendar` class includes a sentence: "Week 1 for a year is the first week that ...", and the document is modified in J2SE 1.3.1: "Week 1 for a year is the earliest seven day period starting on `getFirstDayOfWeek()` that...".

In many cases, a modified document has trivial revisions. While in other cases, revisions are non-trivial, and modified documents may indicate behavioral differences. For example, in ActiveMQ 5.3.0, the document of the `org.apache.activemq.broker.region.cursors.PendingMessageCursor.pageInList(int)` method is "Page in a restricted number of messages". In ActiveMQ 5.3.1, the document is changed to "Page in a restricted number of messages and increment the reference count". Based on the revision, the behavior of the method may change, and the changed method can increase the reference count. We further discuss this issue in Section 3.2.

Syntax 3.57% Library developers may produce a document with syntax errors, and fix them in a later version. For example, in J2SE 1.5, the document of the `java.util.ArrayList.remove(int)` method includes a sentence: "`index` the index of the element to removed". Library developers fix this syntax error in J2SE 1.6, and the modified document is "`index` the index of the element to be removed".

Typo 3.47% Library developers may produce some typos, and fix them in a later version (*e.g.*, possble $\rightarrow$ possible).

Format 0.84% Library developers may modify formats of some words for better presentation. For example, in J2SE 1.5, the document of the `java.util.Vector.setElementAt(E,int)` method includes a sentence: "the set method reverses ...". In J2SE 1.6, library developers change the font of one word, and the modified document is "the `set` method reverses ...". For better readability, a code element within a document is marked by the `@code` annotation in J2SE 1.6.

Implication 2: Many literal polishes such as those for fixing typos, syntax errors, and format issues can be avoided if researchers or practitioners propose an appropriate editor to library developers. It is challenging to implement such an editor for three reasons. (1) Many specialized terms and code names may be detected as typos. For example, although the Eclipse IDE can find some typos, it wrongly identifies "applet" as a typo since it is a specialized term of computer science. (2) Code examples may be detected to include syntax errors. To check these code examples, an editor should understand corresponding programming languages. (3) Specific styles of API documentation may not be well supported by existing editors, and one such style is defined by the official guidance of Java documentation: "library developers should use 'this' instead of 'the' when referring to an object created from the current class".

Finding 3: 22.62% of revisions are about programming tips as follows.

Notice 16.24% Library developers may add notices to describe API usages. Many notices start with the labeling word "Note", but following this style is not a strict requirement. For example, in lucene 3.0.0, library developers add two sentences to the document of the `org.apache.lucene.util.CloseableThreadLocal` class without any labels: "We can not rely on `ThreadLocal.remove()`... You should not call `close` until all threads are done using the instance". In some cases, notices even have no modal verbs such as *must* and *should*. For example, in J2SE 1.5, a notice is added to the document of the `java.util.Observable.deleteObserver(Observer)` method: “Passing `null` to this method will have no effect”.

Library developers may also modify a notice. For example, in J2SE 1.3.1, the document of the `java.util.AbstractCollection.clear()` method has a sentence: “Note that this implementation will throw an `UnsupportedOperationException` if the iterator returned by this collection’s iterator method does not implement the `remove` method”. In J2SE 1.4.2, the modified document includes another condition: “... not implement the `remove` method and this collection is non-empty”. Library developers may even delete notice. For example, in lucene 3.0.0, library developers delete a notice of the `org.apache.lucene.index.IndexWriter.getReader()` method: “You must close the `IndexReader` returned by this method once you are done using it”. It seems that programmers do not have to close the reader explicitly any more.

Code Example 4.36% Library developers may add code examples to illustrate API usages, and later fix defects in code examples. For example, in J2SE 1.5, the document of the `java.util.List.hashCode()` method has a code sample: “`hashCode = 1;`...”, and in J2SE 1.6, a defect is fixed: “`int hashCode = 1;`...”.

As pointed out by Kim *et al.* [12], API documentation in Java typically does not contain as many code examples as API documentation in other languages (*e.g.*, MSDN[9]). Still, library developers of Java libraries are reluctant to add code examples to API documents. Although code examples are useful to programmers, some library developers believe that API documentation should not contain code examples. In particular, the official guidance of Java documentation says “What separates API specifications from a programming guide are examples,...”, so adding many code examples to documentation is against the guidance.

New Description 2.02% Some API elements may not have any documents, or have only automatically generated documents without any true descriptions of usages. In some cases, an API element is found not straightforward to use, so library developers add new descriptions for the API element. For example, in J2SE 1.5, the document of the `java.util.ListResourceBundle.getContents()` method has only one sentence: “See class description”. However, in J2SE 1.6, library developers add detailed explanations to the method: “Returns an array in which each item is a pair of objects in an `Object` array. The first element of each pair is the key, which must be a `String`, and the second element is the value associated with that key. See the class description for details”.

Implication 3: Dekel and Herbsleb [7] show that programming tips such as notices are quite valuable to programmers, but we find that programming tips are challenging to identify since Java does not provide any corresponding annotations. If such annotations are available, tools (*e.g.*, the one proposed by Dekel and Herbsleb [7]) may assist programmers more effectively. In addition, although many programmers complain that API documentation in Java lacks code examples, some library developers are still reluctant to add more code samples partially because doing so violates the principle of writing Java documentation. Some tools (*e.g.*, the tool proposed by Kim *et al.* [12]) may help bridge the gap between programmers and library developers.

[9] `http://msdn.microsoft.com/`

Besides the preceding findings (Findings 1–3), other 2.25% revisions cannot be put into the preceding categories. Some of such revisions are still valuable. For example, in lucene 2.9.1, the document of the `org.apache.lucene.analysis.standard.StandardTokenizer` class includes a sentence that describes a fixed defect and its related bug report: "As of 2.4, Tokens incorrectly identified as acronyms are corrected (see LUCENE-1608)". Tools can use this clue to build relations between bug reports and API code automatically.

3.2 RQ2: To what degree do such revisions indicate behavioral differences?

In this paper, we refer to differences in input/output values, functionalities, and call sequences between two versions of an API library as behavioral differences of API elements. Some behavioral differences can be reflected from revisions of API documentation. In this section, we analyze behavioral differences based on textual revisions of exceptions, parameters, returns, rephrasing, notices, and example code, since these revisions can often indicate behavioral differences as shown in Section 3.1. In total, we find that 18.44% revisions indicate behavioral differences. We classify found behavioral differences into three primary categories, and Figure 3 shows the results. The vertical axis shows the primary categories, and the horizontal axis shows their proportions.

Finding 4: 41.99% of behavioral differences are about exceptions as follows.

Addition 39.19% Library developers may re-implement API methods to throw new exceptions, and add exception documents for these API methods. For example, in J2SE 1.3, library developers re-implement the `java.util.ResourceBundle.getObject(String)` method to throw a new exception (`NullException`), and add a corresponding document. In total, we find that `NullException`, `ClassCastException`, and `IllegalArgumentException` are the top three added exceptions.

Modification 2.04% Library developers may change thrown exceptions of API methods, and modify corresponding documents. For example, in J2SE 1.3, the `Vector.addAll(Collection)` method throws `ArrayIndexOutOfBoundsException`. The thrown exception is changed to `NullPointerException` in J2SE 1.4, and its document is also modified.

Deletion 0.76% Library developers may delete exceptions from API methods, and delete corresponding documents. For example, in ActiveMQ 5.1.0, the `org.apache.activemq.transport.tcp.SslTransportFactory.setKeyAndTrustManagers(KeyManager[],TrustManager[],SecureRandom)` method throws `KeyManagementException`. In ActiveMQ 5.2.0, library developers deprecate this method and delete its exception.

Implication 4: Library developers can add, modify, and delete exception documents, and these modifications indicate behavioral differences. In addition, we find that similar API methods often have similar revisions, and such a similarity could potentially be leveraged to better maintain API documentation.

Finding 5: 29.77% of behavioral differences are about API usage as follows.

Notice 24.68% As discussed in Finding 3, modifications of notices can reflect behavioral differences. For example, in J2SE 1.5, the document of the `java.util.Random.setSeed(long)` method has one notice: "Note: Although the seed value is an AtomicLong, this method must still be synchronized to ensure correct semantics of

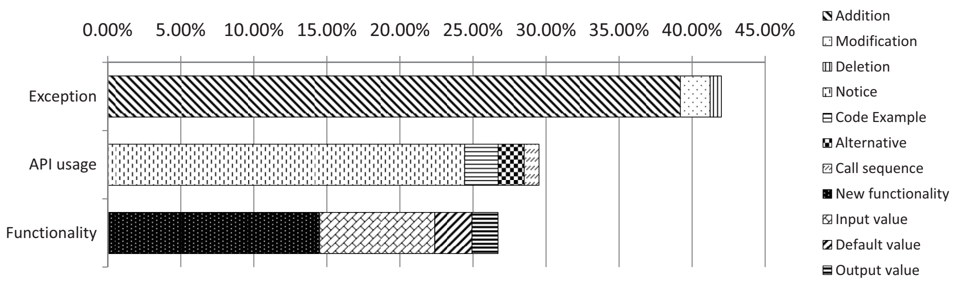


Fig. 3. Categories of behavioral differences.

haveNextNextGaussian". In J2SE 1.6, the notice is deleted. The revision indicates that the latter version does not have to be synchronized as the former version does.

Code example 2.29% Code examples explain API usages, so their revisions very likely reflect behavioral differences. For example, in lucene 2.9.0, the example code of keys in the `org.apache.lucene.search.Hits` class is as follow:

```
TopScoreDocCollector collector = new TopScoreDocCollector(hitsPerPage);
searcher.search(query, collector);
ScoreDoc[] hits = collector.topDocs().scoreDocs;
for (int i = 0; i < hits.length; i++) {
...
```

In lucene 2.9.1, the code example is modified as follows:

```
TopDocs topDocs = searcher.search(query, numHits);
ScoreDoc[] hits = topDocs.scoreDocs;
for (int i = 0; i < hits.length; i++) {
...
```

The version shows that programmers should follow a different way to attain a `ScoreDoc[]` object.

Alternative 1.78% When an API element becomes deprecated, library developers may refer to another API element for the deprecated one as alternatives. Revisions on alternatives very likely reflect behavioral differences. For example, the document of the deprecated `org.apache.lucene.analysis.StopAnalyzer.StopAnalyzer(Set)` constructor is "Use `StopAnalyzer(Set,boolean)` instead" in lucene 2.9.0. In lucene 2.9.1, library developers change the document to "Use `StopAnalyzer(Version,Set)` instead". The revision indicates that another API method should be used to replace the deprecated API method.

Call sequence 1.02% The revisions of API call sequences very likely reflect behavioral differences. For example, in lucene 2.9.2, the document of the `org.apache.lucene.search.Scorer.score()` method has a sentence: "Initially invalid, until `DocIdSetIterator.next()` or `DocIdSetIterator.skipTo(int)` is called the first time". In lucene 3.0.0, library developers modify this sentence: "Initially invalid, until `DocIdSetIterator.nextDoc()` or `DocIdSetIterator.advance(int)` is called the first time". The revision indicates that programmers should call different API methods in the latter version from those in the former version.

Implication 5: As API usages are quite important to programmers, library developers take much effort to improve related documents. However, we find that Java has quite limited annotations to support API-usage documents. If such annotations are available,

Table 2. Percentages of overall API differences

Library	1-2	2-3	3-4	4-5
J2SE	26.05%	59.04%	40.02%	22.60%
ActiveMQ	5.51%	4.95%	5.52%	1.64%
lucene	1.17%	0.51%	13.81%	0.55%
log4j	0.02%	0.00%	13.13%	5.47%
struct	28.17%	8.41%	4.60%	0.20%

library developers can improve readability of API documentation, and programmers can better understand API evolution.

Finding 6: 26.71% of behavioral differences are about functionalities as follows.

New functionality 14.50% Library developers may implement new functionalities for some API elements, and revise corresponding documents. For example, generic is a new functionality introduced in J2SE 1.5, and many related documents are modified (*e.g.*, the document of the `java.util.Collections` class).

Input value 7.89% Library developers may change input ranges of some API methods, and revise corresponding documents. For example, in J2SE 1.3.1, the document of the `java.util.Properties.store(OutputStream,String)` method includes "The ASCII characters \, tab, newline, and carriage return are written as \\, \t, \n, and \r, respectively". In J2SE 1.4.2, library developers add a new ASCII translation for "form feed", and the sentence is modified to "The ASCII characters \, tab, form feed, newline, and carriage return are written as \\, \t, \f \n, and \r, respectively". The revision indicates that the latter version can accept more characters (*e.g.*, form feed) as inputs than the former version does.

Default value 2.54% Library developers may change default values of some API methods, and revise corresponding documents. For example, in J2SE 1.3.1, the document of the `java.util.Hashtable(Map t)` method informs that "The hashtable is created with a capacity of twice the number of entries in the given Map or 11 (whichever is greater)". In J2SE 1.4.2, library developers delete the words about default values, and change the sentence to "The hashtable is created with an initial capacity sufficient to hold the mappings in the given Map". The revision indicates that the capacity of the latter version is different form the former version.

Output value 1.78% Library developers may change output values of some API methods, and revise corresponding documents. In some cases, library developers may find that some output values are not straightforward, so they add documents to these output values. For example, in log4j 1.2.15, the document of the `org.apache.log4j.Appender.getName()` method does not have any sentences about return values. In log4j 1.2.16, library developers add one sentence about return values: "return name, may be `null`". The sentence explains that the return value can be `null`. The revision indicates that the latter version can return `null` values, whereas the former version may not. Still, it is difficult to fully determine whether such a revision indicates behavioral differences or not, and we further discuss the issue in Section 4.

Implication 6: Besides adding new functionalities, library developers also change functionalities by modifying default values and input/output values of some API methods. Although some annotations (*e.g.*, `@param` and `@return`) are provided to describe input and output parameters, value ranges are difficult to identify since they are usually

Table 3. Percentages of API differences by types

(1) J2SE

V	A	R	M
1-2	10.37%	0.61%	89.02%
2-3	15.02%	0.83%	84.15%
3-4	20.12%	4.55%	75.33%
4-5	12.68%	1.40%	85.92%

(2) ActiveMQ

V	A	R	M
1-2	56.80%	5.78%	37.42%
2-3	67.60%	2.23%	30.17%
3-4	42.84%	4.91%	52.25%
4-5	41.61%	7.42%	50.97%

(3) lucene

V	A	R	M
1-2	22.89%	1.21%	75.90%
2-3	7.89%	0.00%	92.11%
3-4	8.71%	41.41%	49.88%
4-5	14.71%	2.94%	82.35%

(4) log4j

V	A	R	M
1-2	50.00%	0.00%	50.00%
2-3	n/a	n/a	n/a
3-4	42.11%	6.57%	51.32%
4-5	38.74%	0.00%	61.26%

(5) struct

V	A	R	M
1-2	26.88%	39.13%	33.99%
2-3	39.13%	12.32%	48.55%
3-4	52.05%	11.06%	36.89%
4-5	0.00%	0.00%	100.00%

mixed with names and descriptions of parameters. In addition, other revisions (*e.g.*, revisions of default values) even have no corresponding annotations. If such annotations are supported, it may be easier to analyze those revisions for behavioral differences.

Besides the preceding findings (Findings 4–6), other 1.53% revisions of behavioral differences cannot be put into the preceding categories.

3.3 RQ3: How frequently are API elements and their documentation changed?

Sections 3.1 and 3.2 have investigated detailed revisions in API documentation. In this section, we present an overall evolution frequency of API differences over versions of all the libraries listed in Table 1. Given two library versions L_1 and L_2, we define the API difference between the two versions of the API library as follows:

$$Dif(L_1, L_2) = \frac{additions + removals + 2 \times modifications}{sum\ of\ public\ elements\ in\ L_1\ and\ L_2} \quad (1)$$

In Equation 1, additions denote newly added API elements; removals denote deleted API elements; and modifications denote API elements whose declarations or documents are modified. We count the number of modifications twice, since each modification can be considered as a removal and an addition.

Finding 7: Contrary to normal expectations, API libraries between two nearby versions are typically quite different. The API differences between two library versions are largely proportional to the differences between the version numbers of the two versions, and *additions* and *modifications* account for most of the proportions by evolution types.

Table 2 shows the overall API differences. Each column denotes the API difference between the two versions of a library. For example, Column “1-2” lists the API differences between a `v1` version and a `v2` version. From Table 2, we find that two versions of a library typically provide different API elements, and only two versions of log4j have exactly the same API elements. For each library, API differences are largely proportional to differences of version numbers. For example, API differences of ActiveMQ are proportional to the differences of its version numbers shown in Table 1.

Table 3 shows the proportions of evolution types. For each library, Column “V” lists versions. For example, Row “1-2” shows the proportions between a `v1` version and a `v2` version. Column “A” denotes proportions of *additions*. Column “R” denotes proportions of *removals*. Column “M” denotes proportions of *modifications*. From the results of Table 3, we find that *additions* and *modifications* account for the most of the proportions of evolution types. It seems that library developers are often reluctant to remove API elements, possibly for the consideration of compatibility across versions.

Implication 7: Based on our results, the API differences between two versions of an API library increase with the differences between their version numbers, so analysis tools for API evolution should deal with more API differences between versions with more different version numbers. Modifications account for the largest proportions. As modifications keep signatures of API methods unchanged, they typically do not cause compilation errors. Thus, analysis tools need to identify and deal with modifications carefully to ensure that the process of API evolution does not introduce new defects into client code.

3.4 Summary

Overall, we find that API documentation between two nearby versions can be quite different, and API differences between versions are proportional to differences of version numbers (Finding 7). Our findings are valuable to better understand API documentation evolution by highlighting the following aspects:

Evolution distribution. Most revisions occur in annotations, but some of annotations are difficult to maintain (Finding 1). Literal polishes account for the second place, and about 30% effort can be saved when an appropriate editor is available (Finding 2). Programming tips account for the third place, and documents on programming tips are challenging to identify since there are no corresponding annotations (Finding 3).

Behavioral differences. Most behavioral differences occur in revisions of exceptions (Finding 4). Although various revisions can indicate behavioral differences, no corresponding annotations exist to support these revisions (Findings 5 and 6).

3.5 Threats to Validity

The threats to external validity include the representativeness of the subjects in true practice. Although we choose five widely used real-world libraries as subjects, our empirical study investigated limited libraries with limited versions, so some findings (*e.g.*, percentages) may not be general. This threat could be reduced by investigating more versions of more libraries. The threats to internal validity include the human factors within our methodology. Although we tried our best to reduce the subjectivity by using double verification, to further reduce this threat, we need to invite more participants to verify our results.

4 Discussion and Future Work

In this section, we discuss issues and our future work.

Variance across version changes. As shown in Table 2, percentages of changes between versions are not fully uniform with variances. To investigate such variances, we plan to use finer-grained analysis in future work. In particular, we plan to investigate the distribution of those variances, their associations, and their styles of common changes for better understanding API evolution.

Determining behavioral differences. It is challenging to automatically determine behavioral differences through only documentation analysis or only code analysis (*e.g.*, code refactoring typically does not cause any behavioral differences). In future work, we plan to combine documentation analysis with code analysis to better determine behavioral differences than with individual techniques.

Benefits of our findings. Our findings are beneficial to programmers, library developers, IDE developers, and researchers. For example, for IDE developers, as our findings reveal that many revisions (*e.g.*, revisions of version numbers) are of little interest, it can be ineffective if IDE developers design an IDE where library developers are required to manually make all types of revisions of API documentation. As another example, for researchers, Mariani *et al.* [14] can also improve their approach that identifies anomalous events, if they consider modified notices and code examples that may lead to anomalous events. Furthermore, we released our results on our project website, so others can analyze benefits of our findings under their contexts.

5 Related Work

Our quantitative study is related to previous work as follows.

Natural language analysis in software engineering. Researchers have proposed approaches to analyze natural language documents in software engineering. Tan *et al.* [21] proposed an approach to infer rules and to detect defects from single sentences of comments. Zhong *et al.* [26] proposed an approach that infers resource specifications from descriptions of multiple methods. Tan *et al.* [22] proposed an approach that infers annotations from both code and comments to detect concurrency defects. Dekel and Herbsleb [7] proposed an approach that pushes rule-containing documents to programmers. Horie and Chiba [11] proposed an extended Javadoc tool that provides new tags to maintain crosscutting concerns in documentation. Buse and Weimer [3, 4] presented various automatic techniques for exception documentation and synthesizing documentation for arbitrary programme differences across versions. Sridhara *et al.* [20] proposed an approach that infers comments of Java methods from API code. Würsch *et al.* [23] proposed an approach that supports programmers with natural language queries. Kof [13] used POS tagging to identify missing objects and actions in requirement documents. Instead of proposing a new approach, we conducted an empirical study that motivates future work on analyzing API documentation in natural languages.

API translation. Researchers have proposed approaches to translate APIs from one API library to another. Henkel and Diwan [10] proposed an approach that captures and replays API refactoring actions to update the client code. Xing and Stroulia [24] proposed an approach that recognizes the changes of APIs by comparing the differences between two versions of libraries. Balaban *et al.* [2] proposed an approach to migrate client code when mapping relations of libraries are available. Dagenais and Robillard [5] proposed an approach that recommends relevant changes of API elements based on comparing API code. Zhong *et al.* [25] proposed an approach that mines API mapping relations for translating APIs in one language to another. Our empirical study reveals various findings and implications on API documentation evolution, and these findings are valuable to improve exiting API translation approaches.

Empirical studies on software evolution or API libraries. Researchers have conducted various empirical studies on software evolution or API libraries. Ruffell and Selby's empirical study [19] reveals that global data is inherent and follows a wave pattern during software evolution. Geiger *et al.*'s empirical study [9] reveals that the relation between code clones and change couplings is statistically unverifiable, although they find many such cases. Bacchelli *et al.*'s empirical study [1] reveals that the discussions of an artifact in email archives and the defects of the artifact are significantly correlated. Novick and Ward's empirical study [16] reveals that many programmers are reluctant to seek help from documentation. Robillard and DeLine [18] conducted an empirical study to understand obstacles to learn APIs, and present many implications to improve API documentation. Padioleau *et al.* [17] presented an empirical study on taxonomies of comments in operating system code. Dagenais and Robillard [6] conducted a qualitative study on creation and evolution of documentation, whereas we conducted a quantitative study on the evolution. Dig and Johnson's empirical studies [8] reveal that refactoring plays an important role in API evolution, and some breaking changes may cause behavioral differences or compilation errors in client code. Our empirical study focuses on the evolution of API documentation, complementing their studies.

6 Conclusion

A quantitative study on API documentation evolution is quite valuable for both programmers and library developers to better understand evolution, and it is difficult to conduct such a study due to various challenges. In this paper, we present an analysis methodology to analyze the evolution of API documentation. We conduct a quantitative study on API documentation evolution of five real-world Java libraries. The results show that API documentation undergoes frequent evolution. Understanding these results helps programmers better learn API documentation evolution, and guides library developers better in maintaining their documentation.

Acknowledgments

We appreciate anonymous reviewers for their supportive and constructive comments. The authors from Chinese Academy of Sciences are sponsored by the National Basic Research Program of China (973) No. 2007CB310802, the Hi-Tech Research and Development Plan of China (863) No. 2007AA010303, the National Natural Science Foundation of China No. 90718042, 60803023, 60873072, and 60903050, and the CAS Innovation Program ISCAS2009-GR. Tao Xie's work is supported in part by NSF grants CNS-0716579, CCF-0725190, CCF-0845272, CCF-0915400, CNS-0958235, an NCSU CACC grant, ARO grant W911NF-08-1-0443, and ARO grant W911NF-08-1-0105 managed by NCSU SOSI.

References

[1] A. Bacchelli, M. DAmbros, and M. Lanza. Are popular classes more defect prone? In *Proc. 13th FASE*, pages 59–73, 2010.

[2] I. Balaban, F. Tip, and R. Fuhrer. Refactoring support for class library migration. In *Proc. 20th OOPSLA*, pages 265–279, 2005.

[3] R. Buse and W. Weimer. Automatic documentation inference for exceptions. In *Proc. ISSTA*, pages 273–282, 2008.

[4] R. Buse and W. Weimer. Automatically documenting program changes. In *Proc. 26th ASE*, pages 33–42, 2010.

[5] B. Dagenais and M. Robillard. Recommending adaptive changes for framework evolution. In *Proc. 30th ICSE*, pages 481–490, 2009.

[6] B. Dagenais and M. P. Robillard. Creating and evolving developer documentation: Understanding the decisions of open source contributors. In *Proc. 18th ESEC/FSE*, pages 127–136, 2010.

[7] U. Dekel and J. D. Herbsleb. Improving API documentation usability with knowledge pushing. In *Proc. 31st ICSE*, pages 320–330, 2009.

[8] D. Dig and R. Johnson. How do APIs evolve? a story of refactoring. *Journal of software maintenance and evolution: Research and Practice*, 18(2):83–107, 2006.

[9] R. Geiger, B. Fluri, H. Gall, and M. Pinzger. Relation of code clones and change couplings. In *Proc. 9th FASE*, pages 411–425, 2006.

[10] J. Henkel and A. Diwan. CatchUp!: capturing and replaying refactorings to support API evolution. In *Proc. 27th ICSE*, pages 274–283, 2005.

[11] M. Horie and S. Chiba. Tool support for crosscutting concerns of API documentation. In *Proc. 8th AOSD*, pages 97–108, 2010.

[12] J. Kim, S. Lee, S. Hwang, and S. Kim. Adding examples into Java documents. In *Proc. 24th ASE*, pages 540–544, 2009.

[13] L. Kof. Scenarios: Identifying missing objects and actions by means of computational linguistics. In *Proc. 15th RE*, pages 121 – 130, 2007.

[14] L. Mariani, F. Pastore, and M. Pezze. A toolset for automated failure analysis. In *Proc. 31st ICSE*, pages 563–566, 2009.

[15] T. Mens and T. Tourwe. A survey of software refactoring. *IEEE Transactions on software engineering*, 30(2):126–139, 2004.

[16] D. G. Novick and K. Ward. Why don't people read the manual? In *Proc. 24th SIGDOC*, pages 11–18, 2006.

[17] Y. Padioleau, L. Tan, and Y. Zhou. Listening to programmers–Taxonomies and characteristics of comments in operating system code. In *Proc. 31st ICSE*, pages 331–341, 2009.

[18] M. P. Robillard and R. DeLine. A field study of API learning obstacles. *Empirical Software Engineering*, to appear, 2011.

[19] F. Ruffell and J. Selby. The pervasiveness of global data in evolving software systems. In *Proc. 9th FASE*, pages 396–410, 2006.

[20] G. Sridhara, E. Hill, D. Muppaneni, L. L. Pollock, and K. Vijay-Shanker. Towards automatically generating summary comments for Java methods. In *Proc. 25th ASE*, pages 43–52, 2010.

[21] L. Tan, D. Yuan, G. Krishna, and Y. Zhou. /* iComment: Bugs or bad comments? */. In *Proc. 21st SOSP*, pages 145–158, 2007.

[22] L. Tan, Y. Zhou, and Y. Padioleau. aComment: Mining annotations from comments and code to detect interrupt-related concurrency bugs. In *Proc. 33rd ICSE, to appear*, 2011.

[23] M. Würsch, G. Ghezzi, G. Reif, and H. Gall. Supporting developers with natural language queries. In *Proc. 32nd ICSE*, pages 165–174, 2010.

[24] Z. Xing and E. Stroulia. API-evolution support with Diff-CatchUp. *IEEE Transactions on Software Engineering*, 33(12):818–836, 2007.

[25] H. Zhong, S. Thummalapenta, T. Xie, L. Zhang, and Q. Wang. Mining API mapping for language migration. In *Proc. 32nd ICSE*, pages 195–204, 2010.

[26] H. Zhong, L. Zhang, T. Xie, and H. Mei. Inferring resource specifications from natural language API documentation. In *Proc. 24th ASE*, pages 307–318, 2009.