Generating Concept based API Element Comparison Using a Knowledge Graph

Yang Liu*
Fudan University
China

Mingwei Liu*
Fudan University
China

Xin Peng*†
Fudan University
China

Christoph Treude
The University of Adelaide
Australia

Zhenchang Xing
Australian National University
Australia

Xiaoxin Zhang*
Fudan University
China

ABSTRACT

Developers are concerned with the comparison of similar APIs in terms of their commonalities and (often subtle) differences. Our empirical study of Stack Overflow questions and API documentation confirms that API comparison questions are common and can often be answered by knowledge contained in API reference documentation. Our study also identifies eight types of API statements that are useful for API comparison. Based on these findings, we propose a knowledge graph based approach APIComp that automatically extracts API knowledge from API reference documentation to support the comparison of a pair of API classes or methods from different aspects. Our approach includes an offline phase for constructing an API knowledge graph, and an online phase for generating an API comparison result for a given pair of API elements. Our evaluation shows that the quality of different kinds of extracted knowledge in the API knowledge graph is generally high. Furthermore, the comparison results generated by APIComp are significantly better than those generated by a baseline approach based on heuristic rules and text similarity, and our generated API comparison results are useful for helping developers in API selection tasks.

CCS CONCEPTS

• **Software and its engineering** → **Documentation**; • **Computing methodologies** → **Information extraction**.

KEYWORDS

API, Knowledge Graph, Documentation, Knowledge Extraction

*Y. Liu, M. Liu, X. Peng, and X. Zhang are with the School of Computer Science and Shanghai Key Laboratory of Data Science, Fudan University, and the Shanghai Institute of Intelligent Electronics & Systems, China.
†X. Peng is the corresponding author (pengxin@fudan.edu.cn).

ASE '20, September 21–25, 2020, Virtual Event, Australia

ACM ISBN 978-1-4503-6768-4/20/09…$15.00
https://doi.org/10.1145/3324884.3416628

ACM Reference Format:
Yang Liu, Mingwei Liu, Xin Peng, Christoph Treude, Zhenchang Xing, and Xiaoxin Zhang. 2020. Generating Concept based API Element Comparison Using a Knowledge Graph. In *35th IEEE/ACM International Conference on Automated Software Engineering (ASE '20), September 21–25, 2020, Virtual Event, Australia.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3324884.3416628

1 INTRODUCTION

Frameworks and libraries often have APIs that provide similar functionalities, but have subtle differences. For example, *java.lang.StringBuffer* and *java.lang.StringBuilder* can be used for string construction, but *StringBuffer* is thread-safe while *StringBuilder* is not. Overlooking such subtle differences between similar APIs may result in program errors, *e.g.,* using *java.lang.StringBuilder* in a multi-thread context. Therefore, developers are often concerned with the comparison of similar APIs. In fact, API comparison questions are common on SO (Stack Overflow). For example, as of March 3, 2019, 13,228 questions tagged with "java" have either the strings "difference between" or "vs" in their title. Among these questions, 38% (5,075 of 13,228) questions do not have an accepted answer.

API reference documentation contains rich knowledge of a variety of aspects of an API, such as functionalities, constraints, directives, caveats, and resource specifications [3, 7, 8, 15, 22, 31, 32]. In an empirical study with 100 JDK API comparison questions from SO, we found that the JDK API reference documentation covers 74% of the points made in the answers to these questions, covering different aspects of API knowledge. We also found that knowledge is scattered within the document of one API element (*e.g.,* class) and across the documents of related API elements, leading to many challenges for API comparison knowledge discovery and summarization. First, API reference documentation has information overloading issues. For example, the API document of *java.nio.file.Files*[1] contains 1,003 sentences. Second, API reference documentation contains diverse types of API knowledge, not all of which are related to API comparison. Third, API reference documentation contains heterogeneous information: code snippets, various aliases (*e.g.,* "string buffer" in the text for *java.lang.StringBuffer*), and co-references (*e.g.,* "this class" may reference different API classes depending on the context).

To assist developers in API selection tasks and automatically generate the comparison of API classes or methods by extracting API

[1]https://docs.oracle.com/javase/8/docs/api/java/nio/file/Files.html

comparison knowledge from API reference documentation, deep understanding of the semantics of the API description text is necessary. Moreover, lots of API knowledge is not only in the text, but also in the code structure, *e.g.,* classes implementing *java.io.Serializable* are serializable. *How can we effectively mine such knowledge from both code and text?* How to normalize and structure the mined API-comparison knowledge is another big challenge, since the same knowledge may be described in different ways in different parts of the API reference documentation. *e.g.,* "A thread-safe, mutable sequence of characters" is the first sentence of *java.lang.StringBuffer* and "A StringBuffer is like a String, but can be modified" is the second sentence, but they describe the overlapping knowledge about *java.lang.StringBuffer*. That is "can be modified" implies the characteristic "mutable". Last but not least, we need a way to automatically infer the commonalities and differences of APIs based on the mined API knowledge to answer API comparison questions.

To tackle these challenges, we propose a knowledge graph based approach APIComp that automatically extracts API comparison knowledge from API reference documentation to support the comparison of a pair of API classes or methods from different aspects (*i.e.,* functionality, characteristic, and categorization). APIComp consists of an offline phase for API knowledge graph construction and an online phase for API comparison service. The offline phase takes as input API reference documentation and produces an API knowledge graph. The online phase generates API comparison results for a given pair of API elements. Our knowledge graph helps to establish extensive relations between API information in different ways, *e.g.,* linking the noun concepts related to APIs to the concepts from a general knowledge graph (*e.g.,* Wikidata [25]). In this way, we can gather API knowledge from different places and in diverse forms, and describe it in a standardized format and present it in an intuitive table for API comparison (see Figure 4).

We evaluated the quality of the key steps for API knowledge graph construction and the effectiveness and usefulness of API comparison results generated by APIComp. Our experimental results show that the quality of different kinds of knowledge in the API knowledge graph is generally high. The comparison results generated by APIComp outperform the comparison results generated by a text similarity based baseline in completeness, conciseness, and understandability while covering more answer points. Moreover, we designed 12 API selection tasks that require API comparison knowledge of similar APIs and asked 12 participants to solve the tasks with APIComp and without APIComp (*i.e.,* using the Google search engine). Our user study shows that participants that used APIComp can solve tasks faster and more accurately compared to those using Google. This shows that APIComp can help developers in real-world API selection tasks. Details of the empirical study and evaluation can be found in our replication package [1].

This paper makes the following contributions:

1) We conducted an empirical study and revealed that API comparison questions can be answered by 8 types of API statements from API reference documentation;

2) We proposed an approach APIComp that automatically extracts API comparison knowledge from API reference documentation to support the comparison of a pair of API elements;

3) We constructed an API knowledge graph for JDK 1.8 (including 188,163 entities and 339,770 relations for 44,809 API classes and methods) and one for Android 27 (including 271,162 entities and 572,098 relations for 77,084 API classes and methods);

4) We evaluated the quality of the key steps for API knowledge graph construction and the effectiveness and usefulness of API comparison results generated by APIComp.

2 EMPIRICAL STUDY

To understand what information developers are looking for when comparing APIs and how we could design an approach to assist developers by providing such comparisons automatically, we conducted an empirical study to investigate whether and where the API reference documentation contains information useful for answering API comparison questions asked on SO. We answered the following research questions:

RQ1: What API comparison information is available on SO?

RQ2: How much useful information does the API reference documentation contain for answering API comparison questions and how scattered is this information in API documentation?

RQ3: What statement types can relevant information for answering API comparison questions be classified into?

2.1 Study Design

2.1.1 Data Preparation. To retrieve questions about API comparison, we selected questions from the SO data dump [20] tagged with "java" that had either of the strings "difference between" or "vs" in the title. We chose Java since the JDK is one of the most popular APIs. We obtained 13,228 such questions. Note that this underestimates the total number of API comparison questions due to our choice of search strings. For this empirical study, we only kept questions with an accepted answer and a score of greater than 10, leading to a total of 1,487 questions. Because we focus on API class/method comparison, we manually removed questions that were not about comparing two JDK API classes/methods by excluding (1) questions aimed at comparing aspects that are not API classes/methods (2) questions involving non-JDK APIs and (3) questions aimed at comparing more than two APIs. The manual removal was conducted by two students independently (one PhD and one MS student, both with more than five years Java experience), with a Cohen's Kappa agreement [10] of 0.897, *i.e.,* almost perfect agreement. We only kept questions that had been annotated as relevant by both students, resulting in 215 questions. Note that the total number of API comparison questions on SO is much higher—the number of 215 is the result of strict filtering (*e.g.,* filtering out all threads with a score $\leq$ 10) to reduce the person power required for manual annotation. To further reduce the effort, we randomly selected 100 API comparison questions out of the 215 questions for subsequent analysis.

2.1.2 Protocol. To answer API comparison questions on SO, users usually summarize information about a certain aspect of the compared APIs (*e.g.,* "Hashtable does not allow null keys or values") or directly compare APIs on a certain aspect (*e.g.,* "Hashtable is synchronized, whereas HashMap is not"). We call this kind of information in SO answers *answer points*. Each answer point can be represented as a sentence and a sentence in an answer may contain multiple answer points. For each of the 100 API comparison questions, we manually extracted answer points from accepted answers, following these criteria:

1) Extraction of answer points must be performed in order from the first sentence of the accepted answer to the last.

2) Answer points must be related to at least one of the two API elements being compared.

3) Extracted answer points must be complete or missing components must be completed, and pronouns must be replaced with the referenced objects.

4) The extracted answer points must be as atomic as possible, describing the knowledge of a single aspect of the API.

Splitting, simplification, completion, and rephrasing of the original sentence are allowed, *e.g.,* two answer points "Hashtable is synchronized" and "HashMap is not synchronized" are extracted from the sentence "Hashtable is synchronized, whereas HashMap is not". If a sentence directly compares two API elements, one answer point is extracted. For example, from "I think the LinkedHashMap has to be faster than HashMap in traversal due to a superior nextEntry implementation in its Iterator.", we will extract "LinkedHashMap is faster than HashMap in traversal", *i.e.,* we make simplifications to the original sentence but retain the basic semantics. This kind of rephrasing has two advantages: (1) to enable better determination of whether the answer point information exists in the API reference documentation (*e.g.,* "Hashtable is synchronized, whereas HashMap is not" may not be described by one sentence in HashMap's reference documentation or HashTable's reference documentation, but each documentation page might describe one half – extracting the original sentence as two answer points makes it easier to find the corresponding information appearing in the API reference documentatio); and (2) to make it easier for us to classify an answer point into a single statement type (cf. RQ3).

For each answer point extracted, we then investigate whether and where the information is available in some form in the API reference documentation by reading the corresponding API reference documentation of JDK 1.8. The documents considered are not limited to the documentation of the API classes and methods being compared, but also include other documents that may be relevant (*e.g.,* documents of parent classes). For investigating where in the API reference documentation the information from an answer point is located, we define the documentation of a class as the whole page of API class documentation including the description of all its members, and the documentation of a method as the description of the method and the entire leading section of the class it belongs to. If more than one page contains the information described by the answer point, we record all pages. Our replication package contains typical examples of extracted answer points and corresponding API documentation.

We conduct the annotation of the 100 questions in two phases: small-scale annotation by two annotators and large-scale annotation by one annotator, following the qualitative research design of previous work (*e.g.,* [4]). During the small-scale annotation, we asked two students (one PhD and one MS student, both with at least five years of Java experience) to extract answer points independently for 20 randomly selected questions. Since different students might use different languages to describe the same answer point, one of the authors examined the answer points extracted to determine whether they were identical (*i.e.,* either used the exact same language or used similar language to describe the same point, *e.g.,* "BufferedReader achieves greater efficiency than InputStreamReader", "BufferedReader is more efficient than InputStreamReader") and to resolve conflicts where needed. Using this protocol, we obtained 54 unique answer points after arbitration, out of which 49 (91%) were extracted by both students. Then for each of the 54 answer points, the same two students investigated whether the corresponding information is available in the API reference documentation, and if so, where. We computed the agreement between the two students for answering whether the information is included in API reference documentation, resulting in a Cohen's Kappa coefficient [10] of 0.764 (*i.e.,* substantial agreement). For the location of the information in API reference documentation, we combined the answers of both students. The remaining 80 API comparison questions were only annotated by one student (*i.e.,* large-scale annotation) following the guidelines summarized from the small-scale annotation.

For answering RQ3, we determined the statement type that the answer points can be classified into by qualitatively analyzing answer points using open coding. The coding was done by three authors of this paper together and started with one seed code (*i.e.,* functionality, the most important knowledge type in API reference documentation [9]). For each answer point, three authors decided which code the answer point can be classified into by discussion. If an answer point could not be classified into an existing code, we created a new code or modified the definition and name of an existing code. If a new code was created or an existing code was modified, we re-annotated all answer points that had been annotated before. We stopped once all answer points had been classified into an existing statement type, *i.e.,* code. To verify that our statement type classification is correct and complete, we invited two MS students (not involved in previous annotation) with more than five years' experience of Java development to use our statement types to annotate all answer points of the 100 questions. The annotation was performed by both students independently and if they thought that an answer point cannot be classified into any existing statement type, they classified it as *unknown*. We analyzed the annotation results and none of answer points was annotated as *unknown* (*i.e.,* no new code is needed) and there are no "not-used" codes (*i.e.,* all codes are useful). The Cohen's Kappa coefficient [10] is 0.880 (*i.e.,* almost perfect agreement). As a result, we consider the statement type classification to be correct and complete.

2.2 Result and Analysis

2.2.1 Answer for RQ1. 255 answer points were extracted from the accepted answers of 100 questions. The most common case is two answer points for one answer (38%), with a maximum of six per answer (one case). 54 questions are about comparing API classes and 46 questions are about comparing API methods.

2.2.2 Answer for RQ2. The information for 189 of the 255 answer points (74%) is available in the API reference documentation. For 85 questions, at least one answer point is available in the API reference documentation. We conclude that most API comparison questions could be completely or partially answered by the API knowledge in API reference documentation.

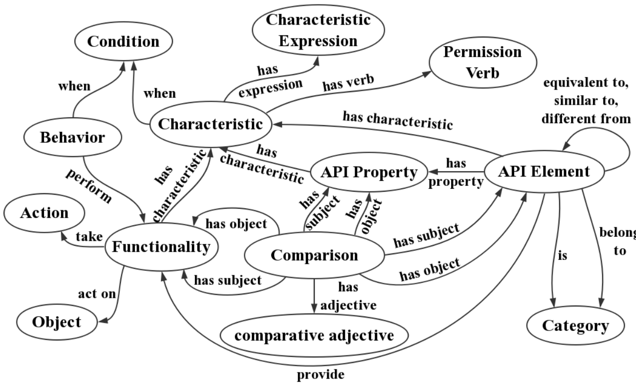


Figure 1: Conceptual Schema of API Statements

For the 85 questions with at least one answer point available in the API reference documentation, we counted how many documents the developer would need to check to answer them. As a result, 20 questions could be answered by only checking one document; but the other 65 questions could only be answered by checking two or more documents (4 at most). In other words, in 76.5% of cases, the information for answering an API comparison question is scattered across documentation of different API elements. This further motivates our work on providing developers with an automated approach for extracting and summarizing API comparison knowledge from API reference documentation.

2.2.3 Answer for RQ3. Table 1 shows the definitions and examples of eight statement types with the number of answer points. Related concepts and their relations can be explained by the conceptual schema shown in Figure 1. These eight statement types are further classified into three aspects: 1) **Categorization**, including concept classification, membership; 2) **Functionality**, including functionality specification, behavior specification, functionality comparison; 3) **Characteristic**, including characteristic specification, characteristic comparison, constraint.

3 APPROACH

The results of the empirical study imply the necessity and possibility of automatically discovering and summarizing API comparison knowledge in the API reference documentation. We can extract relevant API statements and classify them into different types. With the support of relevant knowledge (including concepts and relations) we can align the API statements of two API elements to generate useful API comparison results. We propose a knowledge graph based approach for comparing two API elements. The approach (called APIComp) consists of an offline phase for API knowledge graph construction and an online phase for generating API comparison results (see Figure 2).

API Knowledge Graph Construction. Our API knowledge graph follows the conceptual scheme shown in Figure 1, which is obtained by disassembling the relationships among the subjects, predicates, objects, and conditions involved in the eight types of API statements. We first extract the API structure from the API reference documentation, including API elements (*e.g.,* packages, classes, interfaces, methods) and their relations (*e.g.,* containment, inheritance, implementation). We extract description sentences

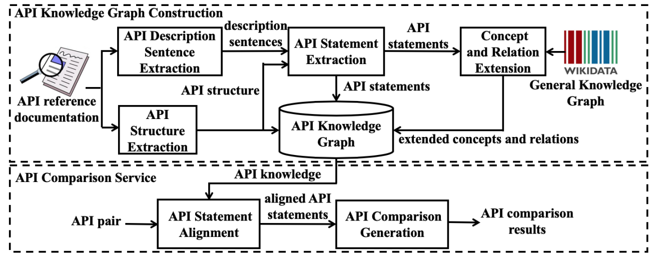


Figure 2: Overview of APIComp

for API elements from the documentation. Based on predefined templates, we use rule-based techniques to extract template normalized API statements from the API structure and API description sentences. The extracted API statements include various concepts (*e.g.,* actions and objects of functionality specifications). To relate API statements to each other and provide concept explanations for them, we further extend the concepts and relations by introducing general concepts that are related to API statements and by identifying additional relations. The general concepts are extracted from general knowledge graphs (*i.e.,* WikiData [25]) and linked with related concepts of API statements. The additional relations are identified between API statements based on both lexical and semantic analysis. The extracted API structure and API statements as well as the extended concepts and relations constitute the API knowledge graph. We describe the details of these steps below.

API Comparison Service. We first align API statements of two given API elements based on the API knowledge graph. The alignment identifies corresponding and comparable API statements for two APIs. The comparison results for the two API elements are generated based on the aligned API statements. The results include a table (see Figure 4) showing the commonalities and differences of the two API elements with explanations for the involved concepts.

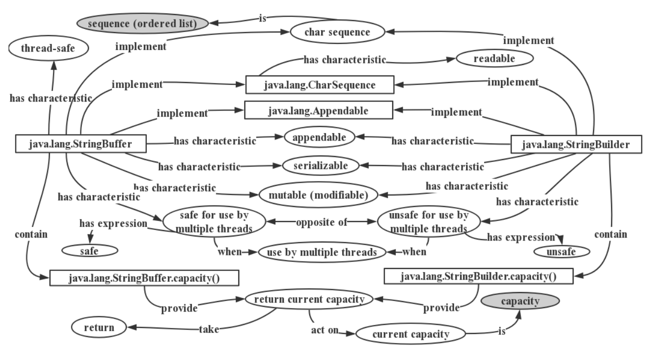


Figure 3: An Example of API Knowledge Graph

3.1 Running Example

Figure 3 shows part of the knowledge graph for the JDK API, where rectangles, white ellipses, and gray ellipses denote API elements, API statements, and extended concepts, respectively. The knowledge graph includes three kinds of knowledge, *i.e.,* API structure, API statements, and extended concepts and relations. The API structure in Figure 3 describes two API classes (*java.lang.StringBuilder* and *java.lang.StringBuffer*), related interfaces and methods, and various relations (*e.g.,* implementation) between them (see Sec. 3.2).

Table 1: API Statement Types Identified in Our Empirical Study

Statement Type	Definition	Example	Class	Method	Total
Concept Classification	Describe that an API element is an instance of a category by concept	PrintWriter is a stream of characters	36	3	39
Membership	Describe that an API element belongs to a category	push operation is part of Stack	1	3	4
Functionality Specification	Describe what an API element can or cannot do	SocketChannel reads from sockets	41	50	91
Behavior Specification	Describe specific behaviors of an API element under a certain condition	FileWriter makes system call when calling to write	8	20	28
Functionality Comparison	Compare the functionalities of two API elements by three relations (equivalent to, similar to, different from)	java.util.Properties is like java.util.Map	6	7	13
Characteristic Specification	Describe the characteristics of an API element, its property (*e.g.,* StringBuffer capacity) or functionality	Hashtable is synchronized	23	13	36
Characteristic Comparison	Compare the characteristics of two API elements, their properties or functionalities	BufferedWriter is more efficient than FileWriter	20	8	28
Constraint	Describe the constraints of an API element or its property using permission verbs (*e.g.,* allow, prohibit, guarantee, limit)	HashSet allows null object	13	3	16
Total			148	107	255

Note: Numbers in the table indicate the number of answer points with corresponding statement type for API comparison questions of classes and methods respectively

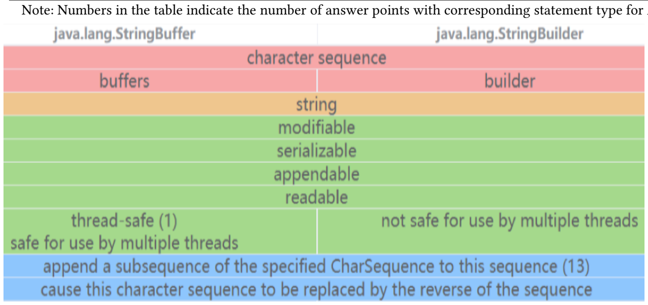


Figure 4: An Example of API Comparison Results

The API statements in Figure 3 describe the categories, functionalities, and characteristics of the two classes, which are extracted from two sources, *i.e.,* API description sentences and API structure (see Sec. 3.4). For example, the characteristic specification "appendable" and the concept classification "char sequence" of the two classes are extracted from their class-interface implementation relations with *java.lang.Appendable* and *java.lang.CharSequence* respectively; the characteristic specifications "thread-safe", "mutable", and "safe for use by multiple threads" of *java.lang.StringBuffer* are extracted from its description sentences "A thread-safe, mutable sequence of characters." and "String buffers are safe for use by multiple threads." from the documentation after sentence completion and API mention resolution (see Sec. 3.3). The extended concepts and relations in Figure 3 conceptually relate API statements (see Sec. 3.5). For example, the relations of opposite characteristic specifications and the shared equivalent characteristic specifications (*e.g.,* "modifiable" and "mutable") are identified to conceptually relate the API statements of the two classes. Moreover, general concepts may also be introduced and linked with the concepts in API statements. *e.g.,* "char sequence" is linked to the WikiData concept "sequence (ordered list)"[2].

Based on the API knowledge graph, APIComp can generate comparison results for any two API elements. For example, the comparison results for *java.lang.StringBuilder* and *java.lang.StringBuffer* are shown in Figure 4 (excerpt), where pink represents concept classification, orange represents membership, green represents characteristic specification, and blue represents functionality specification. To generate the result, we first align the API statements of the two classes (see Sec. 3.6), *e.g.,* "safe for use by multiple threads" and "unsafe for use by multiple threads" are aligned based on their opposite relation and semantic similarity; the category "character sequence", the characteristics "serializable", "appendable" of both classes are aligned based on their equivalence relations. Based on the alignment, a comparison results table is generated by summarizing the commonalities and differences of the two API elements (see Sec. 3.7).

[2]https://www.wikidata.org/wiki/Q133250

3.2 API Structure Extraction

From the API reference documentation we extract four types of API elements, *i.e.,* packages, classes, interfaces, and methods, as well as the following relations between them: **containment** relations between packages, classes/interfaces, and methods; **inheritance** relations between classes/interfaces; and **implementation** relations between classes and interfaces. These API elements and relations can be extracted from the corresponding declarations in the documentation based on their structure.

3.3 API Description Sentence Extraction

To extract description sentences for an API element, we split its text description into sentences. Then we identify and remove sentences that include code statements for reducing noise. To facilitate the extraction of API statements we conduct additional processing, namely sentence completion and API mention resolution, to provide more complete description sentences. After that we filter out short sentences that include no more than two words.

3.3.1 Sentence Completion. The first sentence of the text description of an API element usually provides a brief summary, such as "A thread-safe, mutable sequence of characters." for *java.lang.StringBuffer*. These sentences are often incomplete and lack subjects or predicates. We use an NLP tool (*i.e.,* Spacy) to analyze and identify incomplete sentences based on the following two criteria: it is a declarative sentence; and it has no subject or predicate. For a sentence that has no subject we add the fully-qualified name of the corresponding API element as the subject and if the sentence has no predicate we further add "is" as the predicate.

3.3.2 API Mention Resolution. To facilitate the extraction of API statements we need to replace all the mentions of an API element with its fully qualified name. First we identify all aliases of an API element and replace all occurrences of these aliases in description sentences with the fully qualified name of the corresponding API element. For each API element we recognize the following aliases:

1) the short name (*i.e.,* the part after the last dot of the fully-qualified name) of the API element, *e.g.,* "StringBuilder";

2) the fully-qualified name or short name of the API element (method) without parameters, *e.g.,* "java.lang.StringBuilder.append" and "StringBuilder.append";

3) the phrase obtained by splitting the short name of the API element by camel case and underscore, *e.g.,* "string builder";

4) the phrase obtained by adding the type of the API element (*i.e.,* package, class, interface, or method) after an alias, *e.g.,* "StringBuilder class" and "string builder class".

Then we use a coreference resolution tool (*i.e.,* NeuralCoref[3]) to resolve pronouns which refer to API elements.

3.4 API Statement Extraction

We design a series of heuristic rules to extract API statements from description sentences and the API structure. These rules are summarized by analyzing the description sentences and API structure identified in the empirical study. The word conversion involved in the rules is implemented using WordNet [12].

3.4.1 Extracting from Description Sentences. For each description sentence, we first parse it into simple sentences, then use heuristic rules to extract API statements, and finally normalize the extracted API statements. The process is described below.

We use Spacy to do POS tagging and dependency parsing for the sentence. If the sentence is a compound sentence with multiple predicates, we split it into multiple simple sentences with only one predicate by iteratively executing the following rule based on the dependency tree: for each subordinate clause, if it is an adverbial clause then keep it together with the major clause, otherwise remove it from the sentence, complete its subject if missing, and treat it as a separate sentence. For example, the sentence "java.lang.StringBuffer is like a java.lang.String, but can be modified." will be split into two simple sentences "java.lang.StringBuffer is like a java.lang.String" and "java.lang.StringBuffer can be modified".

Three authors manually analyzed the description sentences from the two packages most involved in the 100 API comparison questions from Sec. 2.1 (*i.e., java.io* and *java.util*), and summarized linguistic patterns iteratively by creating new patterns or modifying and merging existing patterns until all patterns were stable. The resulting 27 linguistic patterns are shown in Table 2. Each linguistic pattern is used as a heuristic rule for API statement extraction. For example, based on the pattern "***AE1*** be [similar as/similar to/like] ***AE2***" (where ***AE1*** and ***AE2*** represent two API elements) we can extract a functionality comparison "similar to *java.lang.String*" for *java.lang.StringBuffer* from the sentence "A StringBuffer is like a String.". Note that multiple API statements of different types may be extracted from a simple sentence using different linguistic patterns. For example, we can extract a category classification "sequence of characters" and two characteristic specifications "thread-safe" and "mutable" for *java.lang.StringBuffer* from "java.lang.StringBuffer is a thread-safe, mutable sequence of characters.".

To facilitate the alignment of API statements we further normalize the phrases in the extracted API statements. First, remove articles at the beginning, such as "a", "an", and "the". Second, for a noun phrase having the form "***NP1*** of ***NP2***", "***NP2***'s ***NP1***, or "***NP2***' ***NP1***", we unify them into the form "***NP2 NP1***". For example, "sequence of characters" will be converted into "character sequence". Third, we convert nouns and verbs to their base forms using WordNet. Fourth, we convert adverbs and passive forms of verbs in characteristic expressions to their adjective forms using WordNet. For example, "can be modified" will be converted into "modifiable".

[3]https://github.com/huggingface/neuralcoref

3.4.2 Extracting from API Structure. The following rules extract API statements from the names of API elements and their inheritance/implementation relations. These rules consider the short names of API elements split by camel case and underscore.

Rule 1: Extracting Functionality Specification from Class-/Interface Name. If the name of a class or interface C includes a noun or noun phrase N_1 followed by another noun N_2 and N_2 can be converted to a verb, then extract a functionality specification for C with the verb form of N_2 as the action and N_1 as the object (*e.g.,* "build string" for *java.lang.StringBuilder*).

Rule 2: Extracting Functionality Specification from Method Name. If the name of a method M includes a verb V followed by a noun or noun phrase N, then extract a functionality specification for M with V as the action and N as the object (*e.g.,* "set length" for *java.lang.StringBuilder.setLength(int)*).

Rule 3: Extracting Characteristic Specification from Class-/Interface Name. If the name of a class or interface C includes adjectives, then for each adjective extract a characteristic specification for C with the adjective as the characteristic expression. (*e.g.,* "writable" for *javafx.scene.image.WritableImage).*

Rule 4: Extracting Characteristic Specification from Inheritance/Implementation Relation. If a class/interface C_1 inherits from or implements another class/interface C_2 and the name of C_2 ends with an adjective, then extract a characteristic specification for C_1 with C_2's name as the characteristic expression (*e.g.,* "serializable" for *java.lang.StringBuilder* from its implementation relation with *java.io.Serializable*).

Rule 5: Extracting Category Classification from Inheritance/Implementation Relation. If a class/interface C_1 inherits from or implements another class/interface C_2 and the name of C_2 is a noun or noun phrase N, then extract a category classification for C_1 with N as the category (*e.g.,* "char sequence" for *java.lang.-StringBuilder* from implements *java.lang.CharSequence*).

3.5 Concept and Relation Extension

Different API statements may use different language to express the same or similar knowledge. To facilitate the alignment of API statements we need to establish conceptual relations between them. In addition, to bridge conceptual gaps we also need to introduce additional concepts and relations from a general knowledge graph.

3.5.1 Equal/Opposite Characteristics. Some API statements describe equal or opposite characteristics of API elements, for example "mutable" and "modifiable" are equal while "safe for use by multiple threads" and "unsafe for use by multiple threads" are opposite. These relations can be discovered by identifying synonyms and antonyms in the adjectives of API characteristics using a lexical database (*e.g.,* WordNet [12]) and thesaurus (*e.g.,* Thesaurus[4]). For two API characteristics AC_1 and AC_2 that have the same conditions or no conditions, we use the following rules to identify possible equal/opposite characteristic relations between them:

1) if the adjectives of AC_1 and AC_2 are synonyms in WordNet or Thesaurus (*e.g.,* "mutable" and "modifiable"), or have the same etymology (*e.g.,* "synchronized" and "synchronous"), add a relation <AC_1, same as, AC_2>;

[4]https://www.thesaurus.com

Table 2: Linguistic Patterns for Extracting API Statements from Description Sentences

Statement Type	Linguistic Pattern	Example
Concept Classification	***AE*** [be/represent] (a/an) JJ* ***NP***	The GridLayout class is a layout manager.
Membership	***[AE/NP]*** [belong to/be part of/be a member of/have] ***[AE/NP]***	Queue is a member of the Java Collections Framework.
Functionality Specification	***AE VB*** ((ADP) ***NP***)+ (RB)	BufferedReader reads text from a character-input stream.
	AE be [used/designed/provided] to ***VB*** ((ADP) ***NP***)+	ClassDesc is used to marshal java.lang.Class objects over IIOP.
	AE be [used/designed/provided] for ***VBG*** ((ADP) ***NP***)+	SynthPainter is used for painting portions of JComponents.
	AE be (JJ/NP) for ***VBG*** ((ADP) ***NP***)+	AsynchronousFileChannel is an asynchronous channel for reading file.
	AE be (JJ/NP) to ***VB*** ((ADP) ***NP***)+	The CertPathBuilder is able to restore prior path validation states.
	NP be ***VBN*** by ***AE***	The modeling of HTML is provided by the class HTMLDocument.
	AE be ***VBN*** ((ADP) ***NP***)+	Image.getSource() is called by the image filtering classes and by methods .
	AE1 VB ((ADP) ***NP***)+ RBR than ***AE2*** (COND)	BufferedWriter writes file faster than OutputStreamWriter.
Behavior Specification	***AE VB*** ((ADP) ***NP***)+ (RB) ***COND***	isCellEditable(EventObject) returns true if anEvent is not a MouseEvent.
	AE be [used/designed/provided] to ***VB*** ((ADP) ***NP***)+ ***COND***	TypeVisitor is used to operate on a type when the kind of type is unknown at compile time.
	AE be [used/designed/provided] for ***VBG*** ((ADP) ***NP***)+ ***COND***	FileReader is used for reading file when IO is ready.
	AE be (JJ/NP) to ***VB*** ((ADP) ***NP***)+ ***COND***	The ImageProducer is free to ignore this call if it cannot resend the data in that order.
	AE be (JJ/NP) for ***VBG*** ((ADP) ***NP***)+ ***COND***	FileInputStream is for reading streams of bytes during threads communicate.
	NP be ***VBN*** by ***AE COND***	File descriptor is modified by FileWriter when the thread starts.
	AE be ***VBN*** ((ADP) ***NP***)+ ***COND***	Object.finalize() is called by the garbage collector on an object when garbage collection determines.
Functionality Comparison	***AE1*** be [same as/equivalent to] ***AE2***	String.copyValueOf(char[]) is equivalent to String.valueOf(char[]).
	AE1 be [similar as/similar to/like] ***AE2***	A StringBuffer is like a String.
	AE1 be [different from/unlike] ***AE2***	InsufficientResourcesException is different from LimitExceededException .
Characteristic Specification	***AE*** be [a/an] ***JJ***+ NP (***COND***)	StringBuilder is a mutable sequence of characters.
	AE be ***JJ*** (***COND***)	Instances of StringBuffer are thread-safe and mutable.
	AE [can/could] be ***VBN*** (***COND***)	StringBuffer could be modified.
	AE VB ((ADP) ***NP***)+ ***RB*** (***COND***)	Filereader reads file efficiently.
Characteristic Comparison	***AE1*** be ***JJR*** than ***AE2*** (***COND***)	ArrayDeque is faster than LinkedList when used as a queue.
	AE1 VB ((ADP) ***NP***)+ ***RBR*** than ***AE2*** (***COND***)	BufferedWriter writes file faster than OutputStreamWriter.
Constraint	***AE PV NP***	IdentityHashMap allows null values and the null key.

Note: AE (API element), NP (noun phrase), VB (verb), ADP (adposition), RB (adverb), RBR (adverb, comparative), JJ (adjective), JJR (adjective, comparative), VBN (past participle), VBG (present participle), PV (permission verb, e.g., allow/guarantee/prohibit/limit), COND (condition, including adverbial clause, prepositional phrase).

2) if the adjectives of AC_1 and AC_2 are antonyms in WordNet or Thesaurus (*e.g.,* "safe" and "dangerous"), or one can be transformed into the other by adding negative prefixes (*e.g.,* "un", "dis", "anti", "ir", "im", "in", "non"), add a relation <AC_1, opposite of, AC_2>.

3.5.2 Noun Concept Categorization. API statements involve many noun concepts, *e.g.,* category in concept classification and membership. The names of these concepts may imply categorization relations, *e.g.,* <buffered writer, is, writer> and <character sequence length, belong to, character sequence>. For two noun concepts C_1 and C_2 in the extracted API statements, we use the following two rules to identify possible categorization relations between them:

1) if C_1's name is shorter than and the prefix of C_2's name and there are no other longer concepts that satisfy this rule for C_1, add a relation <C_2, belong to, C_1>;

2) if C_1's name is shorter than and the suffix of C_2's name and there are no other longer concepts that satisfy this rule for C_1, add a relation <C_2, is, C_1>.

3.5.3 General Concepts and Relations. API statements involve many noun concepts that are included in general knowledge graphs like Wikidata [25]. Relevant concepts and relations in general knowledge graphs provide additional knowledge for API alignment. For example, Wikidata provides knowledge like string is a sequence of characters and a data type, and "str" is an alias of "string". This knowledge not only helps to connect different API statements, but also provides the required concept explanations. To harvest this knowledge we need to link the concepts in API statements to those in Wikidata. This concept linking cannot be easily resolved by name matching, as polysemants are popular among Wikidata concepts. *e.g.,* besides data type, "string" can also be a family name [28], a musical instrument part [29], or a physical phenomenon [30].

To decide whether a concept C_A in API statements can be linked to a concept C_W in Wikidata, we consider: 1) whether the topic of C_W is relevant to the API reference documentation; 2) whether the local contexts of C_A and C_W are similar. We measure both aspects based on the vector representations of words learned using a Word2Vec [11] model. We use the 100-dimensional Word2Vec model pretrained on the Wikipedia corpus[5] and tune the model based on the corpus of all API description sentences using gensim[6]. The topics of the API reference documentation are represented by the names and aliases of all noun concepts involved in API statements. The local context of C_A is reflected by its neighbouring concepts and itself in the API knowledge graph. Similarly the local context of C_W is reflected by its neighbouring concepts and itself in Wikidata.

3.6 API Statement Alignment

Given two API elements, we collect related API statements and identify corresponding statements that can be aligned.

For an API element we collect and consider all its API statements for alignment. If it is a class, we also collect and consider the API statements of the classes/interfaces that it inherits from or implements and the functionality specifications of its member methods. For example, for *java.lang.StringBuffer* we consider the characteristic "readable", as it is the characteristic of the interface *java.lang.CharSequence* which it implements. Then we merge duplicate API statements, *e.g.,* in cases where the same statement was collected from a class and its parent class. To determine whether a statement S_1 of an API element E_1 can be aligned with a statement S_2 of another API element E_2, we calculate their relevance based on both conceptual distance and text similarity.

The conceptual distance between S_1 and S_2 is measured based on their distance in the knowledge graph. Each API statement has a core entity in the knowledge graph as shown in Figure 1: for concept classification or membership, it is the category; for functionality specification, it is the functionality; for behavior specification, it is the behavior; for functionality comparison, it is the other API

[5]https://github.com/3Top/word2vec-api
[6]https://radimrehurek.com/gensim

element in the comparison; for characteristic specification or constraint, it is the characteristic. The distance between S_1 and S_2 can be measured as the length of the shortest path between their core entities in the knowledge graph.

The text similarity between S_1 and S_2 is measured by the similarity of their description words. The description words of an API statement S are the names and aliases of all concepts in $Graph(S)$. We use the same Word2Vec model as in Sec. 3.5.3 to measure the text similarity between S_1 and S_2 by: 1) generating a vector for S_1 and S_2 respectively by averaging the vectors of its description words; and 2) calculating cosine similarity between the two vectors.

Then the conceptual distance and text similarity between S_1 and S_2 can be calculated as Equation 1 and Equation 2 respectively, where $Sim_{cos}(V_{S_1}, V_{S_2})$ is the cosine similarity between the vectors of S_1 and S_2. The combined distance can be calculated as Equation 3, where w_1 and w_2 are two weights satisfying $w_1 + w_2 = 1$. w_1 and w_2 are set to 0.6 and 0.4 respectively by tuning on a test set.

$$Rel_{concept}(S_1, S_2) = 1/(distance(S_1, S_2) + 1) \tag{1}$$

$$Rel_{text}(S_1, S_2) = (Sim_{cos}(V_{S_1}, V_{S_2}) + 1)/2 \tag{2}$$

$$Rel_{combined}(S_1, S_2) = w_1 \times Rel_{concept}(S_1, S_2) + w_2 \times Rel_{text}(S_1, S_2) \tag{3}$$

Finally we determine the alignment between the API statements of E_1 and E_2. First, we generate a set of candidate pairs and each pair has two API statements from E_1 and E_2 respectively. To ensure that only corresponding and comparable API statements are aligned, we divide the API statements into four kinds: 1) concept classification; 2) membership; 3) functionality specification (including its characteristic), behavior specification, functionality comparison; 4) characteristic specification, characteristic comparison, constraint. Only API statements of the same kind can be aligned between two API elements. Second, we remove all candidate pairs whose distance is lower than a threshold (*i.e.,* 0.3 in our implementation). This threshold is set based on preliminary experiments. Third, we consider each of the remaining candidate pairs in the order of relevance (from high to low): if neither of the API statements in the pair is aligned, accept the pair as an aligned pair. Finally, all the accepted pairs are output as the results of alignment (see Figure 4).

3.7 API Comparison Generation

The comparison results between two API elements include three parts: statements for commonalities, statements for differences, and unaligned statements.

For an aligned pair of API statements, if all constituents (*e.g.,* the action and object of a functionality specification, see Figure 1) are the same entities or entities connected by "same as" relations in the knowledge graph, we treat the pair as a commonality; otherwise, we treat it as a difference. Unaligned statements are sometimes duplicated expressions of the same statements. To reduce the duplication we identify and merge duplicated statements of the same API element following the same process as for API statement alignment, *e.g.,* the characteristic "thread-safe" is a duplicated expression of "safe for use by multiple threads" in Figure 4 and merged into latter.

Our concept-based API comparison can further provide explanations for involved concepts (*e.g.,* "thread", "serializable") based on the knowledge graph. The sources of the explanations include the aliases of the concept, the definition of the concept from Wikidata, and the definition of an API element in the documentation.

4 EVALUATION

We constructed an API knowledge graph for JDK 1.8. We developed a web crawler based on Scrapy 1.7.1[7] to obtain HTML pages of the JDK 1.8 API reference documentation[8]. Then we used BeautifulSoup 4.4.0[9] to extract the API structure and text descriptions from the HTML pages. After that we used Spacy 2.1[10] as NLP tool to extract API description sentences and API statements.

The resulting API knowledge graph includes **188,163** entities and **339,770** relations. Among them, there are **44,809** API elements and **52,471** relations between these API elements. The knowledge graph includes **123,627** API statements: **14,336** for concept classification, **21,104** for membership, **62,641** for functionality specification, **14,184** for behavior specification, **705** for functionality comparison, **10,698** for characteristic specification, **394** for characteristic comparison, **270** for constraint. Among these API statements, **22,985** are extracted from API structure and the other **100,642** are extracted from API description sentences. In concept and relation extension, we introduced **2,404** equal/opposite characteristic relations, **117,300** noun concept categorization relations, **6,245** Wikidata concepts and **1,677** noun concept links to Wikidata.

We also applied our approach to Android SDK 27 and obtained the same accuracy and effectiveness results as reported in Section 4.1 and Section 4.2 for JDK. The resulting API knowledge graph includes **271,162** entities and **572,098** relations. Due to the space limitation, we cannot report the experiment results on Android SDK in details in this paper, but all experiments results can be found in the replication package [1].

We conducted a series of experiments to evaluate the quality of the API knowledge and the effectiveness and usefulness of our approach by answering the following research questions:

RQ4 (Quality): What is the intrinsic quality of the knowledge captured in the API knowledge graph?

RQ5 (Effectiveness): How effective is APIComp in generating API comparison results in terms of completeness, conciseness, and understandability?

RQ6 (Usefulness): How useful are the results generated by APIComp in helping developers during API selection tasks?

4.1 Quality of Extracted API Knowledge (RQ4)

Our quality evaluation focuses on API statements as well as extended concepts and relations since the API structure is extracted from structured information and thus intrinsically accurate.

4.1.1 Protocol. Similar to previous studies [7, 26], we adopted a sampling method [18] to ensure that ratios observed in the sample generalize to the population within in a certain confidence interval at a certain confidence level. For a confidence interval of 5 at a 95% confidence level, the required sample size is 384.

We randomly selected 384 API statements for each of the three aspects (*i.e.,* category, functionality, characteristic) and each of the two sources (*i.e.,* API structure, description sentences). For extended concepts and relations, we randomly selected 384 instances for equal/opposite characteristics, noun concept categorization, and

[7] https://scrapy.org
[8] https://docs.oracle.com/javase/8/docs/api
[9] https://www.crummy.com/software/BeautifulSoup/bs4/doc/
[10] https://spacy.io

Table 3: Accuracy of API Statements

Aspect	API Structure		Description Sentences	
	Accuracy	Agreement	Accuracy	Agreement
Functionality	0.820	0.734	0.956	0.850
Category	1.000	1.000	0.956	0.915
Characteristic	0.945	0.975	0.698	0.706

Table 4: Accuracy of Concept and Relation Extension

Extension Approach	Accuracy	Agreement
Equal/Opposite Characteristics	0.740	0.914
Noun Concept Categorization	0.758	0.829
General Concept Linking	0.768	0.779

general concept linking. We invited two Master students (not affiliated to this work) familiar with Java to label the accuracy of the selected samples independently. The criterion is that an extracted API statement or an extended concept/relation is correct and meaningful. They were provided with the sources of the knowledge and related documentation to make decisions. For each sample, if it was labeled differently, a third student was assigned to give an additional label to resolve the conflict by a majority-win strategy.

4.1.2 Results. The results are shown in Table 3 and Table 4. For each sample we provide the accuracy and Cohen's Kappa agreement [10]. We can see the agreement rates are all above 0.7, indicating substantial or almost perfect agreement. The accuracy is generally high (above 0.8) except for the characteristics extracted from description sentences (0.698). We obtained similar results for Android: the accuracy of API statement extraction from API structure and sentences is 0.878-0.940 and 0.682-0.904 respectively; the accuracy of concept and relation extension is 0.706-0.872.

Typical problems of API statements extraction include: 1) incorrect splitting of API names, *e.g.,* "getIssuerX500Principal" is split into "get issuer X 500Principal"; 2) incomplete sentences caused by incorrect HTML parsing or sentence splitting, *e.g.,* "java.sql.ResultSetMetaData.isSearchable(int) indicate."; 3) POS tagging or dependency parsing errors, *e.g.,* "always-on-top" from sentence "java.awt.Window.setAlwaysOnTop(boolean) is always-on-top..." is tagged as a noun; 4) meaningless statements, *e.g.,* "common" is extracted as a characteristic of some APIs. The last one is the primary cause for lower accuracy for extracting characteristics from sentences.

Typical problems of the extension of concepts and relations include: 1) false categorization relation for non-noun concepts, *e.g.,* <second parameter, belong to, second>; 2) false concept linking due to the lack of context, *e.g.,* "accumulator" is linked to "rechargeable battery (accumulator)".

4.1.3 Summary. The quality of different kinds of knowledge (*i.e.,* API structure, statements, and extended concepts/relations) in the API knowledge graph is of high quality. Typical problems with the quality of extracted knowledge include text processing errors, meaningless statements, and false concept linking. These problems can be solved in the future by developing text processing techniques for software text, designing more rules to select meaningful statements, and training models for concept linking and filtering.

4.2 Effectiveness of API Comparison (RQ5)

We compare the API comparison results produced by APIComp to those produced by a baseline approach.

4.2.1 Baseline Approach. Since there is no existing approach that can directly compare two API elements to the best of our knowledge, we implemented a baseline approach based on heuristic rules and text similarity. Given two API elements, we obtain all their description sentences, complete the sentences and resolve API mentions in the same way as the steps described in Sec. 3.3 in our approach. We then select description sentences as the comparison result in the following two ways. First, we select all sentences that mention both API elements (Type 1 sentence). Second, we select sentence pairs that are similar for each API element (Type 2 sentence).

We use the same Word2Vec model (see Sec. 3.5.3) to calculate the similarity between two sentences: 1) remove API elements and convert each sentence to a bag of words after preprocessing (*i.e.,* tokenization, stop word removal, and lemmatization); 2) generate a vector for each sentence by averaging the vectors of their bag of words; and 3) calculate the cosine similarity between the two vectors. We calculate the similarity between each pair of sentences from the two API elements and filter out candidate pairs with low similarity (*i.e.,* lower than 0.6). This threshold is set based on preliminary experiments. Then we order the remaining candidate pairs by similarity and use a greedy selection method to accept pairs from high to low similarity with the condition that none of the sentences in a pair included in an accepted pair. The selected sentences are organized in a table, each row corresponding to a Type 1 sentence or a pair of Type 2 sentences. A screenshot of the baseline is shown in the replication package [1].

This process aims to emulate the process of a developer browsing two pages of the API reference documentation and summarizing the commonalities and differences of two API elements from the documentation, similar to the process suggested by SO questions about comparing API elements (see Sec. 2.1). In contrast, APIComp explicitly constructs a knowledge graph for extracted API statements and identifies corresponding and comparable API statements by combining conceptual distance and text similarity.

4.2.2 Tasks. We randomly selected 20 API comparison questions from the 85 questions in our empirical study that have at least one answer point in the API reference documentation. Each question is used as an API comparison task and only the answer points available in the API reference documentation are considered. In this way, we obtain 20 API comparison tasks with 52 answer points.

4.2.3 Protocol. We invited four Master students (familiar with Java) to evaluate the results. For each task we produced a comparison result by APIComp and the baseline and showed the two results in a random order to the participants. They were asked to evaluate each result in terms of completeness, conciseness, and understandability on a 4-points Likert scale (1-disagree; 2-somewhat disagree; 3-somewhat agree; 4-agree) by the following questions:

1) **Completeness**. Does the result contain all the necessary information to show the commonalities and differences?

2) **Conciseness**. Does the result contain no (or very little) unnecessary or redundant information?

3) **Understandability**. Is the result understandable?

We further conducted a coverage evaluation by comparing the two approaches against the answer points in the corresponding SO questions. We invited two students (one PhD and one Master student) to check the API comparison results independently. For each result they checked each answer point and labeled whether it was covered in the result. If their decisions were different a third

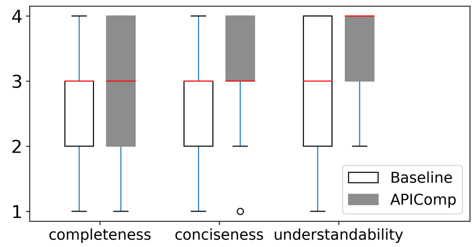


Figure 5: Effectiveness of APIComp and Baseline Approach

student (Master) was assigned to give an additional label to resolve the conflict by a majority-win strategy.

4.2.4 Results. The results of the comparison are shown in Figure 5. For completeness, conciseness, and understandability of APIComp, 63.75%, 83.75%, 92.50% respectively of the answers are 4 or 3 (agree or somewhat agree). For completeness, conciseness, and understandability of the baseline, 58.75%, 58.75%, 67.50% respectively of the answers are 4 or 3 (agree or somewhat agree). Welch's T-test [27] was used for verifying the statistical significance of the difference between the APIComp and baseline ratings for *completeness*, *conciseness*, and *understandability*. The differences are statistically significant ($p << 0.05$) for *conciseness* and *understandability* and not statistically significant ($p = 0.07$) for *completeness*. The coverage evaluation shows that APIComp covers 62.3% of the answer points, while the baseline covers 47.2%. Cohen's Kappa agreement for the two approaches are 0.807 and 0.811 (both almost perfect agreement). We obtained similar results for Android: APIComp covers 19 (79.2%) of the 24 answer points with Cohen's Kappa agreement of 0.864.

The improvement of APIComp over the baseline can be attributed to the knowledge based API statement analysis. For example, for the comparison between *java.util.concurrent.CopyOnWriteArrayList* and *java.util.LinkedList* APIComp can extract a characteristic specification "thread-safe" and "not synchronized" from the sentence "java.util.concurrent.CopyOnWriteArrayList is a thread-safe variant of ArrayList..." and "Note that java.util.LinkedList is not synchronized" respectively. The two API statements are then aligned as a difference based on the "same as" relation between "thread-safe" and "synchronized" and the opposite relation between "synchronized" and "not synchronized". In contrast, the baseline approach aligns the first sentence with another sentence "java.util.LinkedList implements all optional list operations, and permits all elements.".

4.2.5 Summary. Our approach is significantly better than the baseline in terms of *conciseness* and *understandability*. Moreover, our approach covers more answer points of API comparison questions by 15.1 percentage points. The improvement can be attributed to the knowledge graph based API statement analysis.

4.3 Usefulness of API Comparison (RQ6)

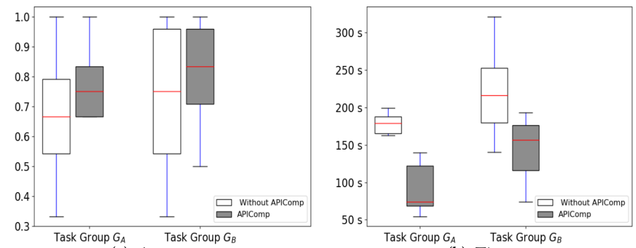
We evaluate the usefulness of APIComp in API selection tasks, that is, choosing the most suitable API element between two API elements in a given scenario. Note that this is different from API retrieval, where the task would be to find potentially suitable API elements among hundreds or thousands of possible elements.

4.3.1 Tasks. We selected API selection tasks from the 215 API comparison questions in our empirical study based on the following criteria: 1) provide a scenario description that can be used to select a single API element from the candidates; 2) have an API element selected in the accepted answer, which indicates that the selected API is the right choice for the given scenario; 3) the API selection can be determined based on the API reference documentation. We ranked the questions meeting the above criteria by their votes and selected Top-6 class comparison questions and Top-6 method comparison questions as the tasks. In this way we got 6 API class selection tasks and 6 API method selection tasks, each with two API elements, a scenario description, and a right answer (*i.e.,* one of the two API elements), all included in the replication package [1].

4.3.2 Protocol. We invited 12 Master students with 1-4 years Java programming experience. They represent novice developers, which are the primary target audience for API comparison. None of them participated in the quality and effectiveness experiments for RQ4 and RQ5. We conducted a pre-experiment survey on their Java programming experience and divided them into two "equivalent" groups (G_A and G_B) based on the survey. We randomly divided the 12 tasks into two groups (T_A and T_B), each with 3 class selection tasks and 3 method selection tasks.

A common way for API selection without APIComp is to use search engines (*i.e.,* Google) to search various Web resources such as API reference documentations, tutorials, and online posts. Therefore, in this experiment we asked participants to complete API selection tasks with APIComp and without APIComp (*i.e.,* only using Google) to evaluate the usefulness of APIComp in API selection tasks. We adopted a balanced treatment distribution for the groups. Participants in group G_A were asked to complete the tasks in group T_A with APIComp and the tasks in group T_B without APIComp. Conversely, participants in group G_B were asked to complete the tasks in group T_B with APIComp and the tasks in group T_A without APIComp. For each participant, the tasks were interleaved, one completed with APIComp and one without APIComp. For each task, a participant was asked to select an API element from two candidates for a given scenario description. If participants completed tasks without APIComp, they can search with any keywords on Google and check any Web pages except the corresponding SO question. The participants using APIComp make the decision based on only the results generated by APIComp. A participant can submit one of the two candidate API elements as the answer or none of them if he/she cannot determine. The correctness and completion time of each participant for each task were recorded.

4.3.3 Results. Figure 6 shows the accuracy (*i.e.,* the ratio that the right APIs were selected by a participant group for a task) and the completion time of the two participant groups over the two groups of tasks when completed with APIComp and without APIComp respectively. Using APIComp participants (in both groups) completed the tasks 41% faster (82 seconds on average) and 14.5% more accurately (about 0.10) than without APIComp. We use Welch's T-test for verifying the statistical significance of the differences. The difference in time is statistically significant ($p << 0.05$), while the difference in accuracy is not significant ($p = 0.18$).



(a) Accuracy (b) Time

Figure 6: Usefulness Evaluation for API Selection Tasks

Note that without APIComp the participants can search on Google not only the API reference documentation but also other online resources (*e.g.,* blogs) that discuss an API selection task. *e.g.,* the API elements compared in the task "Which class is more efficient for non-threaded applications? java.util.Hashtable or java.util.HashMap" are often discussed together. It is therefore easy for the participants to find the right answer from Google search results. The API elements in another task "When developing a JDBC driver, which one should be used if considering the exception chaining mechanism? java.lang.Throwable.getCause() or java.sql.SQLException.getNextException()." are not often discussed together. For this task the participants chose the right API element much faster (74s vs 200s) and more accurately (0.83 vs 0.67) with APIComp.

4.3.4 Summary. Our approach significantly decreases the amount of time developers need for API selection tasks. The advantage is more significant when the compared API elements are not often discussed together online.

5 THREATS TO VALIDITY

The empirical study and the evaluation share common threats to validity. A threat to the internal validity is the subjective judgment in different parts, for example the evaluation of the quality of extracted API knowledge. To alleviate this threat we have reported the agreement for each subjective judgment or the corresponding statistical significance. A threat to the external validity is the limited number of subjects (*e.g.,* API comparison questions, tasks) considered in different parts and the fact that we only consider JDK and Android APIs. Our findings may not generalize to other libraries. Another threat to the internal validity of the evaluation is the baseline approach used in the effectiveness study (see Sec. 4.2) which was implemented by ourselves and may not be optimized. To alleviate this threat we have tried to follow state-of-the-art techniques (*e.g.,* Word2Vec) to create a comparable tool.

6 RELATED WORK

API documentation is an important source of knowledge for software developers, leading to a substantial body of work on API documentation. Shi *et al.* [17] conducted a quantitative study of API documentation evolution and found that it undergoes frequent evolution. Monperrus *et al.* [13] presented a study on directives in API documentation and a taxonomy of 23 kinds of API directives. Maalej and Robillard [9] reported on a study of knowledge patterns in API documentation, such as functionality, concepts, and directives. They found that most API comparison questions could be answered with knowledge from the API reference documentation. In this work, we further classify the statements used to answer API comparison question into 3 aspects and 8 statement types.

Other work related to API documentation has attempted to enrich API documentation with other sources, *e.g.,* by recovering traceability links between APIs and their learning resources [2], discovering relevant tutorial fragments [6], linking source code examples to API documentation [21], or extracting API-related insights from Stack Overflow [24]. These approaches link APIs with relevant text or code fragments in various learning resources, but they do not deeply mine the knowledge that already exists in the API documentation. In contrast, we extract API statements from API reference documentation and store them as a knowledge graph. Further, we help to answer API comparison questions from Stack Overflow with API documentation, which is a supplement to previous work [24].

Other researchers have attempted to extract useful pieces of knowledge from API documentation by inferring API specifications and directives such as resource specifications [31], method specifications [15], and parameter constraints and exception-throwing declarations [32], or API caveats [7]. These types of knowledge are useful for understanding the usage of APIs, in particular in terms of API directives. In contrast, we focus on extracting API statements related to three aspects (functionality, characteristic, and categorization) which are relevant to API comparison.

There are also many studies for document comparison generation [5, 16, 23] for other domains (*e.g.,* news reports). These cannot be applied to API comparison since they (1) do not take into account the specific types of knowledge required for API comparison; (2) are designed for documents with other characteristics, *e.g.,* without code elements; and (3) cannot mine knowledge from the API structure which is essential for API comparison.

Other work focuses on generating summaries for API elements. Sridhara *et al.* [19] generated summaries for Java methods using structure and linguistic information. Moreno *et al.* [14] provided JSummarizer to automatically generate summaries of Java classes, and Liu *et al.* [8] designed *KG-APISumm* to generate query-specific API class summaries through an API knowledge graph constructed from API reference documentation. All of these can only generate summaries for a single API element and the information contained in their summaries is not applicable to API comparison involving two API elements.

7 CONCLUSION

In this paper, we conducted an empirical study on API comparison questions and identified 8 types of API statements that are useful for API comparison. We proposed a knowledge graph based approach APIComp for generating API comparison results. Our evaluation confirms the quality of various kinds of knowledge in the knowledge graph, and the effectiveness and usefulness of the generated API comparison results. In the future, we will improve and extend our approach by supporting context aware API comparison and automatically identifying and recommending similar API elements.

ACKNOWLEDGMENTS

This work is supported by National Natural Science Foundation of China under Grant No. 61972098.

REFERENCES

[1] 2020. *Replication Package*. Retrieved August 31, 2020 from https://fudanselab.github.io/Research-ASE2020-APIComp/

[2] Barthélémy Dagenais and Martin P. Robillard. 2012. Recovering traceability links between an API and its learning resources. In *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*. 47–57.

[3] Davide Fucci, Alireza Mollaalizadehbahnemiri, and Walid Maalej. 2019. On using machine learning to identify knowledge in API reference documentation. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*, Marlon Dumas, Dietmar Pfahl, Sven Apel, and Alessandra Russo (Eds.). ACM, 109–119.

[4] Hideaki Hata, Christoph Treude, Raula Gaikovina Kula, and Takashi Ishio. 2019. 9.6 million links in source code comments: purpose, evolution, and decay. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*. 1211–1221.

[5] Xiaojiang Huang, Xiaojun Wan, and Jianguo Xiao. 2011. Comparative News Summarization Using Linear Programming. In *The 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies, Proceedings of the Conference, 19-24 June, 2011, Portland, Oregon, USA - Short Papers*. The Association for Computer Linguistics, 648–653.

[6] He Jiang, Jingxuan Zhang, Zhilei Ren, and Tao Zhang. 2017. An unsupervised approach for discovering relevant tutorial fragments for APIs. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*. 38–48.

[7] Hongwei Li, Sirui Li, Jiamou Sun, Zhenchang Xing, Xin Peng, Mingwei Liu, and Xuejiao Zhao. 2018. Improving API Caveats Accessibility by Mining API Caveats Knowledge Graph. In *2018 IEEE International Conference on Software Maintenance and Evolution, ICSME 2018, Madrid, Spain, September 23-29, 2018*. 183–193.

[8] Mingwei Liu, Xin Peng, Andrian Marcus, Zhenchang Xing, Wenkai Xie, Shuangshuang Xing, and Yang Liu. 2019. Generating query-specific class API summaries. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*. 120–130.

[9] Walid Maalej and Martin P. Robillard. 2013. Patterns of Knowledge in API Reference Documentation. *IEEE Trans. Software Eng.* 39, 9 (2013), 1264–1282.

[10] Mary L McHugh. 2012. Interrater reliability: the kappa statistic. *Biochemia medica: Biochemia medica* 22, 3 (2012), 276–282.

[11] Tomas Mikolov, Ilya Sutskever, Kai Chen, Gregory S. Corrado, and Jeffrey Dean. 2013. Distributed Representations of Words and Phrases and their Compositionality. In *Advances in Neural Information Processing Systems 26: 27th Annual Conference on Neural Information Processing Systems 2013. Proceedings of a meeting held December 5-8, 2013, Lake Tahoe, Nevada, United States*. 3111–3119.

[12] George A. Miller. 1995. WordNet: A Lexical Database for English. *Commun. ACM* 38, 11 (1995), 39–41.

[13] Martin Monperrus, Michael Eichberg, Elif Tekes, and Mira Mezini. 2012. What should developers be aware of? An empirical study on the directives of API documentation. *Empirical Software Engineering* 17, 6 (2012), 703–737.

[14] Laura Moreno, Andrian Marcus, Lori L. Pollock, and K. Vijay-Shanker. 2013. JSummarizer: An automatic generator of natural language summaries for Java classes. In *IEEE 21st International Conference on Program Comprehension, ICPC 2013, San Francisco, CA, USA, 20-21 May, 2013*. 230–232.

[15] Rahul Pandita, Xusheng Xiao, Hao Zhong, Tao Xie, Stephen Oney, and Amit M. Paradkar. 2012. Inferring method specifications from natural language API descriptions. In *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*. 815–825.

[16] Xiang Ren, Yuanhua Lv, Kuansan Wang, and Jiawei Han. 2017. Comparative Document Analysis for Large Text Corpora. In *Proceedings of the Tenth ACM International Conference on Web Search and Data Mining, WSDM 2017, Cambridge, United Kingdom, February 6-10, 2017*, Maarten de Rijke, Milad Shokouhi, Andrew Tomkins, and Min Zhang (Eds.). ACM, 325–334.

[17] Lin Shi, Hao Zhong, Tao Xie, and Mingshu Li. 2011. An Empirical Study on Evolution of API Documentation. In *Fundamental Approaches to Software Engineering - 14th International Conference, FASE 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings*. 416–431.

[18] Ravindra Singh and Naurang Singh Mangat. 2013. *Elements of survey sampling*. Vol. 15. Springer Science & Business Media.

[19] Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori L. Pollock, and K. Vijay-Shanker. 2010. Towards automatically generating summary comments for Java methods. In *ASE 2010, 25th IEEE/ACM International Conference on Automated Software Engineering, Antwerp, Belgium, September 20-24, 2010*, Charles Pecheur, Jamie Andrews, and Elisabetta Di Nitto (Eds.). ACM, 43–52.

[20] StackOverflow. 2019. Stack Overflow data dump version from March 3, 2019. https://archive.org/download/stackexchange/.

[21] Siddharth Subramanian, Laura Inozemtseva, and Reid Holmes. 2014. Live API documentation. In *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*. 643–652.

[22] Jiamou Sun, Zhenchang Xing, Rui Chu, Heilai Bai, Jinshui Wang, and Xin Peng. 2019. Know-How in Programming Tasks: From Textual Tutorials to Task-Oriented Knowledge Graph. In *2019 IEEE International Conference on Software Maintenance and Evolution, ICSME 2019, Cleveland, OH, USA, September 29 - October 4, 2019*. IEEE, 257–268.

[23] Maksim Tkachenko and Hady W. Lauw. 2019. CompareLDA: A Topic Model for Document Comparison. In *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, The Thirty-First Innovative Applications of Artificial Intelligence Conference, IAAI 2019, The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019*. 7112–7119.

[24] Christoph Treude and Martin P. Robillard. 2016. Augmenting API documentation with insights from stack overflow. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*. 392–403.

[25] Denny Vrandecic. 2013. The Rise of Wikidata. *IEEE Intelligent Systems* 28, 4 (2013), 90–95.

[26] Chong Wang, Xin Peng, Mingwei Liu, Zhenchang Xing, Xuefang Bai, Bing Xie, and Tuo Wang. 2019. A Learning-Based Approach for Automatic Construction of Domain Glossary from Source Code and Documentation. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 97–108.

[27] Bernard L Welch. 1947. The generalization of Student's problem when several different population variances are involved. *Biometrika* 34, 1/2 (1947), 28–35.

[28] wikidata. 2019. String. https://www.wikidata.org/wiki/Q37484380.

[29] wikidata. 2019. string. https://www.wikidata.org/wiki/Q326426.

[30] wikidata. 2019. string. https://www.wikidata.org/wiki/Q1376436.

[31] Hao Zhong, Lu Zhang, Tao Xie, and Hong Mei. 2011. Inferring specifications for resources from natural language API documentation. *Autom. Softw. Eng.* 18, 3-4 (2011), 227–261.

[32] Yu Zhou, Ruihang Gu, Taolue Chen, Zhiqiu Huang, Sebastiano Panichella, and Harald C. Gall. 2017. Analyzing APIs documentation and code to detect directive defects. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*. 27–37.