Mining API Usage Examples from Test Code

Zixiao Zhu[1, 2], Yanzhen Zou[1, 2], Bing Xie[1, 2, †], Yong Jin[1, 2], Zeqi Lin[1, 2], Lu Zhang[1, 2]
[1] Software Institute, School of Electronics Engineering and Computer Science, Peking University
[2] Key Laboratory of High Confidence Software Technologies, Ministry of Education
Beijing, 100871, China
{zhuzx11, zouyz, xiebing, jinyong11, linzq14, zhanglu}@sei.pku.edu.cn

***Abstract*—Lack of effective usage examples in API documents has been proven to be a great obstacle to API learning. To deal with this issue, several approaches have been proposed to automatically extract usage examples from client code or related web pages, which are unfortunately not available for newly released API libraries. In this paper, we propose a novel approach to mining API usage examples from test code. Although test code can be a good source of usage examples, the issue of multiple test scenarios might lead to repetitive and interdependent API usages in a test method, which make it complicated and difficult to extract API usage examples. To address this issue, we study the *JUnit* test code and summarize a set of test code patterns. We employ a code pattern based heuristic slicing approach to separate test scenarios into code examples. Then we cluster the similar usage examples for recommendation. An evaluation on four open source software libraries demonstrates that the accuracy of our approach is much higher than the state-of-art approach *eXoaDoc* on test code. Furthermore, we have developed an Eclipse plugin tool *UsETeC*.**

Keywords—API; usage example; test code; code patterns; code slicing

I. Introduction

Many studies [1], [2], [3] have confirmed that usage examples can be a key resource for learning and reusing Application Programming Interfaces (APIs). Unfortunately, many software libraries fail to provide sufficient usage examples in official documentation [5], [6].

To make up for the lack of efficient usage examples in API documents, quite a few approaches have been proposed to automatically extract API usage examples from existing resources. There are generally two sources for extracting API usage examples. The first one is the source code of client projects that contain real usages of the APIs [5], [6], [7], [8], [9], [10], [11], [12]. The second source is the web pages that contain discussion of some usage issues of the APIs [13], [14], [15]. However, neither client code nor related web pages are available for a newly released API library. The existing approaches based on these two sources can hardly work for the APIs which are new or not yet widely used.

For new APIs, test code could be a good source for extracting usage examples. We can see some inherent advantages of test code. First, unit tests target the designed functionality of the APIs, in which the developers are most interested. Second, test code is generally fine-grained and executable. It is more suitable to produce correct and self-contained code examples. Third, test code is available when software libraries are newly released. Nasehi et al. [19] have conducted a study to investigate the suitability of unit tests as a source of API examples. Their qualitative result reveals that unit tests are very helpful, but finding the right usage example from test code might be tricky.

We conducted a pilot study on test code. We found that it is far from easy to migrate existing example mining approaches from client code to test code. First, a test method is usually a mixture of multiple test scenarios. Each test scenario indicates a complete usage of some API methods. Different scenarios in a test method usually focus on the same API usage and should be separated properly. Second, a test method generally has an API method under test while using some helper API methods. Considering the roles of the invoked API methods, the mined code examples might be more representative for the target API method than the helper methods. However these are not considered in existing code mining approaches.

To address these problems, we propose a novel approach to automatically mining API usage examples from test code. Based on the pilot study, we summarize a set of code patterns of unit tests and propose a heuristic slicing based approach, which can be used to separate multiple test scenarios in test code and extract candidate code examples. For the extracted candidate code examples, we adapt the clustering technique to assemble examples of the similar API usage. Finally, we have developed an Eclipse plugin tool named *UsETeC* (<u>Us</u>age <u>E</u>xamples from <u>Te</u>st <u>C</u>ode). *UsETeC* provides two ways of recommending the usage examples to users.

We evaluate our approach by comparing with a state-of-art approach *eXoaDoc* [5], [6]. Our experimental results on four open source projects demonstrate that the average accuracy of our approach is about 40% higher than the *eXoaDoc* approach. Furthermore, we compare our mined examples with official Javadoc examples and have proven the usefulness and code quality of our result.

To the best of our knowledge, our approach is the first to mine API usage examples from test code. More specifically, the main contributions of this paper include:

- A set of test code patterns that characterizes the test methods containing multiple test scenarios.
- A novel approach to mining API usage examples from test code, which integrates the heuristic scenario separation strategies with the code slicing algorithm.
- A prototype tool (*UsETeC*) implementing our approach.

[†] Corresponding Author.

The paper is organized as follows. Section II explains the technical challenges. Section III describes the test code patterns. Section IV presents our approach. Section V presents our evaluation. Section VI reviews related research work. Section VII makes the conclusion.

II. Challenges

A ***test scenario*** in unit tests usually contains three phases: data preparation, test execution and result verification. In *JUnit*, result verification is implemented by *assertion* methods. The testing engine will run the unit test, and judge whether it passes or fails the test according to the return values of assertion methods. Also developers can easily infer the expected program state by reading the assertion methods (such as Line 12-13 in Fig. 1). These characteristics of test code make the mined usage examples more readable to understand the API designment and rationale.

```
1    public void testKeySetByValue() {
2       BinaryTree m = new BinaryTree();
3       LocalTestNode nodes[] = makeLocalNodes();
4       Collection c1 = new LinkedList();
     ...
5       m = BinaryTree.initial();
6       c1.clear();
7       for (int k = 0; k < nodes.length; k++){
8          m.put(nodes[ k ].getKey(), nodes[ k ]);
9          if (k % 2 == 1)
10            c1.add(nodes[ k ].getKey());
11      }
12      assertTrue(m.keySetByValue().retainAll(c1));
13      assertEquals(nodes.length / 2, m.size());
     ...
14      m = BinaryTree.initial();
15      c1.clear();
16      for (int k = 0; k < nodes.length; k++){
17         m.put(nodes[ k ].getKey(), nodes[ k ]);
18         if (k % 2 == 0)
19            c1.add(nodes[ k ].getKey());
20      }
21      assertTrue(m.keySetByValue().removeAll(c1));
22      assertEquals(nodes.length / 2, m.size());
     ...
23   }
```

Fig. 1. A test method with multiple test scenarios.

A major obstacle to extracting API examples from test code is the multiple test scenarios in a test method. Fig. 1 depicts such a test method. Lines 2-4 are the declaration of some data objects. Lines 5-13 depict a test scenario that contains the usage of some API methods, such as *keySetByValue*, *put*, and *getKey*. Lines 14-22 depict another test scenario, which contains a similar usage to the previous one. Such multiple test scenarios are quite reasonable when aiming at covering testing input domains. But they bring redundant code for API users to read. In fact, there are actually 200+ code lines containing similar test scenarios in the test method in Fig. 1. It is necessary to separate different test scenarios from one test method and cluster the similar usages to remove redundancy.

Unfortunately, existing approaches might fail to solve this problem. We can see that the two scenarios in Fig. 1. share the same data objects, the same data preparation procedures (lines 2-4) and are executed sequentially. The code extraction module in existing approaches is based on data-flow analysis [6], [12] or control-flow analysis [7], [10], [11]. It could hardly distinguish different test scenarios, because the scenarios are syntactically dependent on each other.

III. Test Code Patterns

We have conducted a pilot study on two Java libraries, Commons-Math[1] and HttpClient[2]. The goals of our pilot study are: 1) gaining an overview of the multiple test scenarios issue; 2) investigating the characteristics of test code; 3) seeking a solution for separating test scenarios. Through the study, we summarize a set of test code patterns. For each pattern, we propose a set of heuristic rules for separating test scenarios.

A. Pilot Study

1) Procedure

We randomly sample 36 classes and 15 classes from two subjects, Commons-Math and HttpClient, respectively. That is approximately 10% of the test classes in the two subjects. All the test methods (with *JUnit* annotation *@Test*) in the 51 classes were thoroughly inspected by three authors. We manually counted the following indicators:

A. Number of test scenarios in a test method. We have tried to separate the scenarios manually.

B. Number of shared data objects among test scenarios in a method, i.e., the same data objects are used by different scenarios.

C. Number of test execution statements in a test scenario.

D. Whether the test scenarios in a method have the similar test execution phase (i.e., method invocation sequences are the same)?

E. Number of assertion statements in a scenario.

2) Results

TABLE I. depicts the statistics of multiple scenarios in test methods. There are 51.9% (232/447) of the test methods containing more than one test scenario. A test method contains 3.6 test scenarios on average. This number will increase to 6.0 if we only concern the methods with multiple test scenarios. According to the last column in TABLE I. , a test scenario contains 1.3 assertion statements on average. These statistics confirm that multiple test scenarios is a widespread phenomenon in test code.

TABLE I. Pilot Study: Multiple Scenarios in Test Methods

Subject	**# of Test Methods**		**Average Scenarios**		**# of Assertion per Scenario**
	Single Scenario	***Multiple Scenarios***	***in All Methods***	***in Multi-Scenario Methods***	
Commons-Math	170	169	3.0	5.0	1.2
HttpClient	45	63	5.4	8.6	1.3
Total	215	232	3.6	6.0	1.3

[1] http://commons.apache.org/proper/commons-Math/, accessed in Mar. 2013.
[2] http://hc.apache.org/httpcomponents-client-ga/, accessed in Mar. 2013.

According to the result of indicators B, C, D, we separate the 232 test methods containing multiple test scenarios into five categories.

- **C1**: In 40 test methods, test scenarios do not share data objects and do not contain any test execution statements.
- **C2**: In 50 test methods, test scenarios share data objects but do not contain any test execution statements.
- **C3**: In 58 test methods, test scenarios share data objects and have similar test execution phase, i.e., method invocations sequences of the scenarios are same.
- **C4**: In 75 test methods, test scenarios share data objects and have different test execution phases.
- **C5**: In 9 test methods, test scenarios do not share data objects and contain at least one test execution statement.

Base on the first four categories, we summarize four code patterns. The category C5 is an unusual case in test methods, and traditional code slicing algorithms would work well for such methods. So we do not need to take this category into acount when summarizing code patterns.

In the following sections, we describe the definition, examples and scenario separating strategy for each code pattern. Before further discussion, we first introduce our categorization of code statements, which is used in defining code patterns. We give a brief description and some examples for the three statement categories in TABLE II.

B. Code Pattern #1

Pattern Definition: A test method belonging to code pattern #1 contains only assertion statements, such as the method *testConstructor* in Fig. 2. Each assertion statement contains the three phases of a test scenario. Such a scenario indicates very simple and basic API usages. This pattern is based on test method category C1.

```
public void testConstructor() {
   Assert.assertEquals(1.625, new BigReal(new
      BigDecimal("1.625")).doubleValue(), 1.0e-15);
   Assert.assertEquals(-5.0, new BigReal(new
      BigInteger("-5")).doubleValue(), 1.0e-15);
   Assert.assertEquals(-5.0, new BigReal(new
      BigInteger("125"), 3).doubleValue(), 1.0e-15);
}
```

Fig. 2. An example of code pattern #1.

We use an extended regular expression technique to help understand code patterns. The character alphabet of our regular expression is the labels of statement categories (denoted by abbreviations). The language recognized by a regular expression is a set of statement sequences that match the specified code pattern. For code pattern #1, the regular expression is:

$$<Asrt>*$$

It means that the statement sequences of this pattern are a repetition of *Test Assertion* statements.

Scenario Separating Strategy: It is the easiest situation that separating the statements one by one will suffice.

C. Code Pattern #2

Pattern Definition: A test method belonging to code pattern #2 contains consecutive assertion statements and a few non-assertion statements. Fig. 3 shows an example. The non-assertion statements include: necessary data declarations (line 2) and re-initialization of the variables (line 5). Line 3, 4, 6, 7 each indicates a test scenario. They share the data object *array*, but the method under test *WordUtils.initials()* is invoked inside the assertion statement and there is no leading test execution statement for each scenario. This pattern is based on test method category C2.The regular expression for pattern #2 is:

$$<Nasrt>*(<Nasrt><Asrt>|<Asrt>)*$$

```
1    public void testInitials_String_charArray() {
2      char[] array = null;
3      assertEquals("S", WordUtils.initials("SJC", array));
4      assertEquals("BJ", WordUtils.initials("Ben J.Lee",
          array));
       ...
5      array = "SIJo1".toCharArray();
6      assertEquals("C", WordUtils.initials("SJC", array));
7      assertEquals("B.", WordUtils.initials("Ben J.Lee",
          array));
       ...
8    }
```

Fig. 3. An example of code pattern #2.

The sub-expression before the parenthesis represents the statements of data setup at the very beginning of the test method. The remaining part of the expression represents the consecutive assertion statements occasionally leading by some data re-initializtaion statements.

Scenario Separating Strategy: The following strategy works for code pattern #2.

Step 1: Identify every assertion statement.

Step 2: For each assertion statement, find the variables used in the statement. Slice out the declaration and the last value-modifying statement of every used variable.

TABLE II. CATEGORIES OF STATEMENTS IN TEST CODE.

Categories	Abbr.	Definition	Examples
Target Invocation	*Tgt*	At least one invoked method in the statement is proven to be the API method under test.	`array = set.getCharRanges();` `// ` *getCharRanges* is the method under test.
Test Assertion	*Asrt*	The statement that contains the invocations of *assertion* methods. The assertion methods usually start with some special words like "assert", "test", "check", "fail", etc.	`assertEquals(array[index], set.getCharAt(i));`
Non-assertion Statement	*Nasrt*	The statements that are not *Test Assertion* statements.	`m = BinaryTree.initial();` `c1.clear();`

D. Code Pattern #3

Pattern Definition: A test method belonging to code pattern #3 consists of test scenarios that each contains at least an invocation of the API method under test. This pattern is based on test method category C3. We have slightly changed the restriction of C3: focused on the Target Invocation and relaxed the restriction of method invocation sequences. This is for ease of pattern identification and scenario separation. The regular expression for code pattern #3 is:

(<Nasrt>*<Tgt><Nasrt>*<Asrt>+)*

The sub-expression inside the parentheses represents a test scenario. The sub-expression <Nasrt>*<Tgt><Nasrt>* describes a sequence of non-assertion statements that contain at least one target invocation statement, which corresponds to the data preparation and test execution phases in a test scenario. The sub-expression *<Asrt>+* describes consecutive test assertion statements in the end of a scenario. The outermost asterisk symbol (*) indicates that such a scenario might repeat many times in a method.

Scenario Separating Strategy: We use the following strategy for code pattern #3.

Step 1: Separate test code into groups. A code group starts after the last assertion statement in the previous group and stops before the first non-assertion statement in the next group.

Step 2: We use a code slicing algorithm described in Fig. 4 to pick out the data-relevant statements in each code groups. A data-relevant statement is the statement that defines or modifies the data objects used in the target invocation statement.

Step 3: Extract the declaration statements and last modifying statements of all the variables used in the data-relevant statements produced by step 2. These statements together with the data-relevant statements form a complete scenario.

```
Algorithm slicing: Slice data-relevant statements in code group
Input: Code group G. List of target invocations in G, denoted by T.
Output: List of data-relevant statements S.
Procedure:
1   let $S \leftarrow \varnothing$
2   let $V \leftarrow \varnothing$
3   for each statement s in T do
4       $S \leftarrow S \cup \{s\}$
5       $R \leftarrow getUseVariables(s)$
6       for each variable v in R do
7           $V \leftarrow V \cup \{v\}$
8   for each statement s' in G in reverse order do
9       $R' \leftarrow getDefVariables(s')$
10      if $R' \cap V \neq \varnothing$ then
11          $S \leftarrow S \cup \{s'\}$
12          for each variable v' in R' do
13              $V \leftarrow V \cup \{v'\}$
14  Output: S
```

Fig. 4. Pseudo-code of slicing algorithm.

E. Code Pattern #4

Pattern Definition: A test method belonging to code pattern #4 consists of test scenarios that have common sub sequences of method invocations. This pattern is based on test method category C4. We have also changed the restriction of C4: requirement of common sub sequences would help to separate the scenarios more accurately. Fig. 5 shows a typical test method of this pattern. The method tests a set of basic functionality of API class *BasicAuthCache*, including the method *put*, *get*, *remove* and *clear*. There are three test scenarios in the method: line 4-5, line 6-7, line 8-10. They share two data objects, *cache* and *authScheme*. Their method invocation sequences are not same and there is no unified test target method. But there is a common subsequence among three method invocation sequences, i.e., the invocations of *get* and *HttpHost*.

```
1    public void testBasics() throws Exception {
2       final BasicAuthCache cache = new BasicAuthCache();
3       final AuthScheme authScheme =
                             Mockito.mock(AuthScheme.class);
4       cache.put(new HttpHost("localhost", 80), authScheme);
5       Assert.assertSame(authScheme, cache.get(
                             new HttpHost("localhost", 80)));
6       cache.remove(new HttpHost("localhost", 80));
7       Assert.assertNull(cache.get(
                             new HttpHost("localhost", 80)));
8       cache.put(new HttpHost("localhost", 80), authScheme);
9       cache.clear();
10      Assert.assertNull(cache.get(
                             new HttpHost("localhost", 80)));
11   }
```

Fig. 5. An example of code pattern #4.

It is hard to summarize an accurate regular expression of this pattern. We analyze the method invocation sequences to recognize and slice test methods of this pattern.

Scenario Separating Strategy: The pattern classification and scenario separation for this pattern is combined in one process. The algorithm is described as follows.

Step 1: Separate the test method into groups by the assertion statements and generate the sequence of method invocations for each code group.

Step 2: We use the longest common subsequence (LCS) algorithm to find the LCS among the method invocation sequences of different code groups. If no valid LCS is found, this method does not belong to pattern #4.

Step 3: Execute the slicing algorithm in Fig. 4 for each code group. We use the method invocations in LCS as the second input parameter of the algorithm.

Step 4: Extract data declaration and last modifying statements for the relevant variables in the output statements of step 3.

IV. USETEC

Based on the four code patterns, we propose our approach to mining API usage examples from test code. The approach consists of three main steps. It firstly identifies the API method under test in a test method, then a pattern-based heuristic slicing approach are proposed to extract code examples from test methods. Finally, the extracted usage examples are clustered for recommendation. As a result, we have developed an Eclipse plugin tool, named *UsETeC* (Usage Examples from Test Code).

A. *Identifying API Methods Under Test*

We retrieve the information of API methods under test to use in separating test scenarios and recommending examples.

Naming Convention (NC) has been proven as one of the most accurate strategy for identifying the class under test in test code [28]. Developers usually name the test methods after the name of the API method under test. For example, the name of test method in Fig. 1 is *testKeySetByValue*, while the method under test is *keySetByValue*.

We compare the text similarity between test method name and the invoked API method name. First, we split the camel-cased method names into words. Then we perform Porter's algorithm to stem those split words. Given the stemmed word sets of a test method name and the API method name, we count the number of common words in two sets and calculate the name similarity using the following formula.

$$\text{Name Similarity} = \frac{2 * |Common\ Words|}{|Words\ of\ Test\ Method\ Name| + |Words\ of\ API\ Method\ Name|} \quad (1)$$

The symbol |*Common Words*| denotes the number of words in the set of common words. The other two symbols are in the same way. The similarity metric is virtually the harmonic mean of the ratios of common words in the two word sets. We select the invoked API method with the highest name similarity with the test method as the method under test. If none of the invoked API methods has any common words with the test methods, the name similarities are all zero. In this case, the NC strategy cannot find out the API method under test.

B. *Extracting Candidate Code Examples*

We propose a heuristic slicing approach for extracting code examples from test methods, which is based on the four code patterns. The approach contains two main steps: code pattern classification and test scenario separation.

1) *Code Pattern Classification*

For each input test method, we first decide which code pattern it belongs to. We generate the category sequence of statements in a test method, and use the regular expressions of code patterns to recognize the sequences. If one regular expression recognizes the sequence, the test method would fall into the corresponding code pattern. The recognization for code pattern #4 is based on the method invocation sequence, as described in section III.

Because the four code patterns are not orthogonal, the priority order of code patterns would be an important factor here. We have considered this problem when designing the patterns. There is an inherent partial order between the four patterns: #1 < #2 < #3 < #4. The partial order < means *is-a-special-form-of.* This partial order is easy to infer from our pattern definition. So the appropriate priority order of pattern classification is code pattern #1, #2, #3, #4, from first to last.

2) *Test Scenario Separation*

After identifying the code pattern for a test method, we use the corresponding scenario separation algorithm to process the method. The algorithms is based on the strategies for each code pattern, which is described in section III.

There are some test methods not belonging to any of the four code patterns. In such case, a baseline algorithm is invoked. In the baseline algorithm, we implemented the code slicing algorithm proposed by *eXoaDoc*, and adapted it to test code by separating scenarios by assertion statements.

C. *Clustering Similar API Usage Examples*

The goal of clustering is to assemble the examples of similar API usages and remove redundancy. The clustering algorithm is described in Fig. 6.

The similarity between two examples is evaluated with a metric used in code clone detection [24]. We extract the sequence of method invocations for every usage example. The example similarity is calculated by the formula (2).

$$Similarity\ (Example\ e1,\ Example\ e2) = \frac{2 * |LCS(S(e1),\ S(e2))|}{|S(e1)| + |S(e2)|} \quad (2)$$

Here the symbol |s($e1$)| and |s($e2$)| denotes the number of method invocations in example $e1$, e2, respectively. The symbol |LCS(s($e1$), s($e2$))| denotes the number of method invocations in the longest common subsequence of the method invocation sequences of the two methods. This metric is also a harmonic mean.

The clustering algorithm relies on a parameter, namely, *threshold*. We set it to 0.5, i.e., approximately half of the invocations in the two sequences are the same. The optimization of the chosen threshold is subject of future work.

We select one or more representative examples from each cluster to form the final example dataset. We calculate an example's average similarity with all the other examples in a cluster. The representative example should have the highest average similarity in a cluster.

```
Algorithm clustering: Clustering usage examples
Input: A set of usage examples, denoted by S.
Output: Clusters of usage examples, denoted by C.
Procedure:
S' ← S;
Step 1:
    Initiate a new cluster c
    Randomly select a usage example x from S' as the clustering seed of c
Step 2:
    for each example s in S' - x
        Calculate similarity(x, s)
        if similarity(x, s) > threshold
            Add s into c
            Remove s from S'
Step 3:
    Add c into C
    if S' ≠ ∅
        Repeat step1-3
```

Fig. 6. Clustering algorithm.

D. *UsETeC Tool: Recommending Examples*

The *UsETeC* tool collects all the mined API usage examples, and organizes the examples by the related API methods. For each usage example, the related API methods are the test targets in the test method from which the example is mined. If we can not find the test target or the target method is not inovked in the example, we relate the example to all the invoked API methods in it. Ranking the examples is subject of our future work.

UsETeC provides two ways to present and recommend examples. The first way is the automatically generated Javadoc, with the usage examples embedded into the documentation paragraphs of the related API methods. The second way is a customized way that the developers can choose the examples to add into Javadoc, through the human-interactive plugin GUI integrated with the Eclipse IDE.

We have applied *UsETeC* on four open source projects (i.e., the subjects used in our experimental evaluation). Due to space limit, the demonstration of the tool usage is not included in this paper. The tool and the related resources, including the user manual, the generated Javadoc and some indicative examples are available at: http://tsr.sei.pku.edu.cn/usetec/index.html.

V. Experimental Evaluation

In our evaluation, we mainly investigated the following three research questions.

RQ1: How effective is our approach to mine correct API usage examples from test code?

RQ2: How many API methods in new libraries can our approach produce usage examples for?

RQ3: Do the usage examples mined by our approach show the typical API usages?

The three research questions are concerned with the effectiveness, applicability and usefulness of our approach, respectively. The first research question is concerned with the effectiveness of our approach to mine correct usage examples. The second question is concerned with whether our approach can produce sufficient examples for the new APIs. The third question is concerned with whether the mined usage examples are useful for learning the APIs.

A. *Experimental Design*

We used four open-source Java projects as the subjects: Commons-Lang[1], Commons-Math[2], JFreeChart[3], and Apache POI[4]. The reasons for choosing these Java libraries are as follows. First, these four projects cover different domains of application. Second, the test code of these projects is rich and well-organized. Third, participants in our manual analysis phase are familiar with these libraries to guarantee the quality of our golden standard dataset. TABLE III. depicts the basic information of each subject.

[1] http://commons.apache.org/proper/commons-Lang/, accessed in Mar. 2013.
[2] http://commons.apache.org/proper/commons-Math/, accessed in Mar. 2013.
[3] http://www.jfree.org/jfreechart/, accessed in Mar. 2013.
[4] http://poi.apache.org/, accessed in Mar. 2013.

TABLE III. Basic information of the subjects

Projects	API Methods	Test Code		
	Num.	*kLOC*	*Classes*	*Test Methods*
Commons-Lang	1346	44	116	1704
Commons-Math	2645	81	356	2920
JFreeChart	5909	91	390	2981
Apache POI	5092	81	447	2293
Total	**14992**	**297**	**1309**	**9898**

First, we generated a raw dataset by sampling about 200 test methods from each subject. The raw dataset consists of 808 test methods. Note that, we excluded the methods that have been inspected in the pilot study.

Second, we manually built up a *Golden Standard Dataset* (*GSD* for short) as the standard answers of extracting API usage examples from test methods. We employed 15 Java programmers who have 3-7 years of Java programming experience to manually inspect the dataset and give the golden standard. A three-round double-checked process was used.

- In the first round, each analyst was asked to manually separate the test scenarios in a test method by picking out the statements that form a usage example.
- In the second round, we exchanged the sub-dataset for every analyst and repeated the process of first round. Each test method was thus analyzed by at least two analysts.
- In the third round, the conflicts in the previous two rounds were handled. If two analysts of a test method could not reach an agreement, two experienced analysts, who have 7 years of Java programming experience, joined the discussion and made the decision.

We generated 2493 standard API usage examples for 808 test methods in the raw dataset. They formed the *GSD*.

Third, we applied our *UsETeC* tool and the re-implemented *eXoaDoc* tool to the raw dataset. We choose *eXoaDoc* as comparing work because it is the state-of-art work on mining API usage examples from client code, and we used its code slicing algorithm as a baseline algorithm in our approach.

Finally, we compared the results of the two tools using the *GSD*. To quantitatively evaluate the correctness of extracted examples, we leveraged the metric of edit distance to measure the similarity between mined example and the standard example.

B. *RQ1-Effectiveness*

1) *Accuracy Metrics*

Edit distance is a metric for measuring the similarity between two strings or sequences. The *Levenshtein distance* between two sequences is the minimum number of editing operations (including insertion, deletion, or substitution) to transform one into the other [30]. The edit distance between two code snippets can be calculated by counting the statement-based editing operations to transform one snippet to another.

TABLE IV. OVERALL EFFECTIVENESS OF OUR APPROACH (USETEC) AND EXOADOC.

Approach	**Subject**	**Percentage of the *GSD* examples with**					**Average *min-ed***
		min-ed = 0	***min-ed <= 1***	***min-ed <= 2***	***min-ed <= 4***	***min-ed <= 10***	
UsETeC	Commons-Lang	57.84%	68.87%	76.51%	92.92%	99.60%	1.29
	Commons-Math	36.29%	42.10%	50.65%	79.03%	94.68%	2.33
	JFreeChart	24.15%	33.57%	58.94%	73.67%	93.96%	3.26
	Apache POI	43.10%	50.86%	53.45%	63.79%	93.97%	3.19
	Total	***45.72%***	**54.95%**	**65.65%**	**84.58%**	**97.07%**	**1.99**
eXoaDoc	Commons-Lang	1.83%	7.48%	16.00%	33.04%	60.91%	6.90
	Commons-Math	0.96%	4.82%	12.36%	22.15%	44.78%	4.51
	JFreeChart	3.37%	6.01%	12.74%	27.16%	53.61%	6.25
	Apache POI	3.13%	7.03%	14.06%	32.81%	50.78%	6.78
	Total	**1.94%**	**6.52%**	**14.40%**	**29.22%**	**55.96%**	**6.17**

For each standard usage example in *GSD*, we find out the most similar example in the mined result set. We refer to the edit distance between the *GSD* example and its most similar example as the ***minimum edit distance*** (***min-ed*** for short) of the *GSD* example in the specific result set. When the *min-ed* of a *GSD* example in a result set is 0, it means that the result set contains an identical example with the *GSD* one. We use the **average *min-ed*** of all the *GSD* examples in a result set to measure the quality of the result set. If the average *min-ed* is smaller, the overall accuracy of the result set is better, and thus the approach that producing the result set is more correct and effective.

2) Results Analysis

TABLE IV. depicts the effectiveness of *UsETeC* and the *eXoaDoc* approach. In this table, columns 3-7 depict the percentage of *GSD* examples whose *min-ed*s are in the five ranges depicted in the table (i.e. *min-ed*=0, <=1, <=2, <=4, <=10). Each row depicts the statistics for different subjects and different result sets. For example, the italic and underscored cell of row 7, column 3 depicts that in the overall result set of *UsETeC*, the *min-eds* of 45.72% *GSD* examples equal zero. In other word, our approach extract identical examples with 45.72% *GSD* examples. The last column depicts the average *min-ed* of the *GSD* for each subject in each result set. From TABLE IV. , we have the following observations.

For each subject and in each *min-ed* range, our *UsETeC* approach consistently achieves a far higher *GSD* example percentage than *eXoaDoc*. For each subject, the average *min-ed* of *GSD* in the *eXoaDoc* result set is more than two times larger than that in the *UsETeC* result set. These observations mean that the accuracy of *UsETeC* examples is better than that of *eXoaDoc* examples.

We further demonstrate the discrete distribution of the *min-ed*s of the *GSD* examples in the two result sets, as in Fig. 7. From this chart, we have the following observations. First, the distribution of the *GSD min-ed* in the *UsETeC* result set concentrates on the low-value range, mostly under 5. On the contrary, the distribution in the *eXoaDoc* result set concentrate on the high-value range, mostly above 10. Second, when the *min-ed* is smaller, the gap between two approaches is larger. We can interpret that our approach often produce very similar or even identical examples with *GSD* examples, while the *eXoaDoc* approach often fail to provide accurate examples for the usages indicated by *GSD* examples.

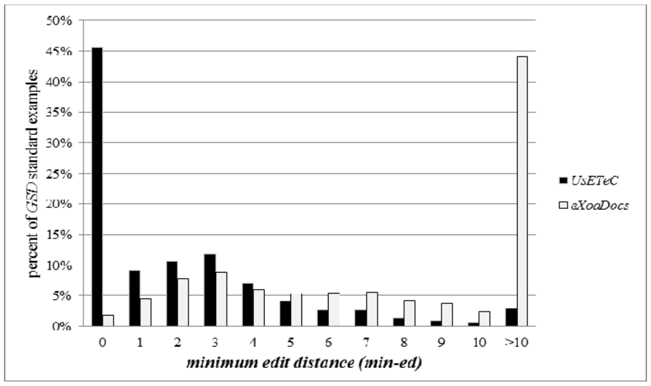

Fig. 7. Distribution of the *min-ed*s of *GSD* examples in the two result sets.

3) Contribution of Different Code Patterns

As our approach employs four code patterns, we also investigated how each pattern contributes to our approach. TABLE V. depicts the applicability of each code pattern for each subject. For example, the underscored cell of row 6, column 4 depicts that code pattern #3 has an overall 46.48% applicability on the test methods. That is to say, 46.48% of the test methods are classified as pattern #3 and handled by the slicing algorithm for pattern #3. We observed that code pattern #3 and #4 have the highest applicability, about 68% of the test methods are classified as these two patterns. Besides, 18.57% of the test methods fall into none of the code patterns.

TABLE V. APPLICABILITY OF THE CODE PATTERNS.

Subject	**Pattern #1**	**Pattern #2**	**Pattern #3**	**Pattern #4**	**Default**
Commons-Lang	23.20%	5.20%	57.60%	0.00%	14.00%
Commons-Math	6.90%	2.96%	49.75%	24.14%	16.26%
JFreeChart	0.49%	0.98%	47.06%	21.08%	30.39%
Apache POI	4.79%	6.59%	25.15%	35.33%	28.14%
Overall Result	**9.83%**	**3.88%**	***46.48%***	**21.24%**	**18.57%**

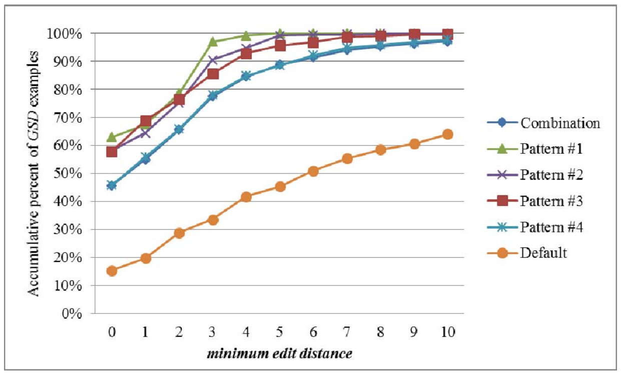

Fig. 8. Effectiveness of different code patterns.

Fig. 8 depicts the effectiveness of different patterns. Each curve indicates the accuracy of the specific pattern's scenario separation strategy when applying to the test methods belonging to the specific pattern. A point of coordinate (x, y%) means that in the sub-dataset processed by the specific code pattern, the min-eds of y% *GSD* examples are less than x. We observed that the most effective code pattern is pattern #1, followed by pattern #2, pattern #3 and pattern #4.

C. *RQ2-Applicability*

TABLE VI. depicts the example coverage for the API methods in the four subjects. From this table, we have the following observations. First, for each subject, our approach provides examples for 30% to 50% more APIs than official Javadoc. Second, the applicability of our approach differs obviously among the four subjects, but the lowest example coverage is still higher than the highest example coverage in Javadoc.

Base on the observations, we can conclude that our approach can mine a significant number of usage examples for new APIs, but the quantities may vary among projects. The design of the API libraries (e.g., the principle of 80/20) might affect the example coverage of our approach.

TABLE VI. MINED EXAMPLE COVERAGE OF API METHODS IN LIBRARIES.

Project	**No. of API methods**	**API methods with examples in official Javadoc**		**API methods with examples mined by our approach**	
		Num.	***Perc.***	***Num.***	***Perc.***
Commons-Lang	1346	391	29.05%	843	62.63%
Commons-Math	2645	21	0.79%	1337	50.55%
JFreeChart	5909	0	0.00%	1897	32.10%
Apache POI	5092	14	0.27%	1928	37.86%
Total	14992	426	**2.84%**	6005	**40.05%**
Average per project		-	**7.42%**	-	**45.79%**

D. *RQ3-Usefulness*

In order to evaluate the usefulness of our examples, we conducted a human study of our examples in comparison with the human-writtern examples in official Javadoc. As shown in TABLE VI. , there are 426 API methods enriched with examples in Javadoc, mostly from the Commons-Lang project. We used the Javadoc examples to evaluate the value of mined examples, base on a hypothesis that the human-written examples are of high quality and show the best practice of the APIs.

1) *Experimental Procedure*

First, we added all the API examples in the official Javadoc of the four subjects into a benchmark dataset.

Second, we picked out all the *UsETeC* usage examples mined for the 426 API methods.

Finally, we invited 10 experienced Java programmers to make an evaluation. For each API method involved in the benchmark dataset, the participant was shown the Javadoc examples and mined examples at the same time. The participants were requested to analyze the method invocation sequence of the mined and Javadoc examples and make the following judgment. For each Javadoc example in benchmark dataset, *UsETeC* has mined:

a) The Identical Example. There are at least one mined example having the same method invocation sequence as the Javadoc example.

b) A Similar Example. Some mined examples invoke the same API methods as the Javadoc example, but the method invocation sequence are not completely identical.

c) Nothing. Our approach failed to mine any examples for the same API method indicated by the Javadoc examples.

2) *Results*

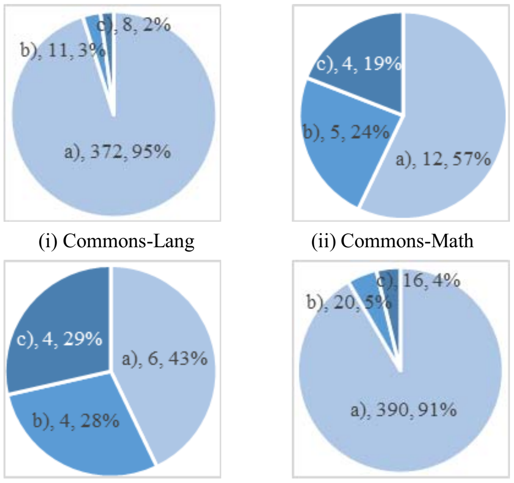

Fig. 9. Comparison between mined examples and Javadoc examples

Fig. 9 demonstrates the statistics for each subject and the overall result (we excluded JFreeChart because no API examples found in its Javadoc). The data labels on the pie chart depict the chosen option, the number and percentage of the benchmark API methods of that option.

From Fig. 9, we have the following observations. For each subject, the situation of *UsETeC could mine the identical* *example with benchmark API example* is the most common. As for the overall result, our approach mined identical examples for 91% of the benchmark API methods. We can conclude that the usage examples mined by our approach cover most API usages recommended in the official documentation.

Following the quantitative analysis, we present an instance for further discussion. In Fig. 10, the two usage examples illustrate the usages of the API method *lookup* of the API class *StrLookup*. Example I in the upper cell is written in official Javadoc, example II in the lower cell is mined from test code by *UsETeC*. The two examples both depict the usage of looking up the value in the map data structure by a given key. We can infer that the two examples focus on the same usage of the same API method.

From Fig. 10, we have some interesting observations. The first is that line 2-3 of example I are identical with line 3 and line 5 of example II. The second is that the Javadoc example contains the invocation of method *assertEquals*, which is used only in test code. Base on the observations, we can make a reasonable conjecture that the authors of API documentation and the developers of test code might learn from each other's ideas or even copy-paste code fragments. The fact that the API designers and developers believe that test code snippets demonstrate the right API usages would provide strong supporting evidence for the value of mined usage examples in API learning.

```
1    Map map = new HashMap();
2    map.put("number", Integer.valueOf(2));
3    assertEquals("2",
     StrLookup.mapLookup(map).lookup("number"));
```

Example I. The usage example in Javadoc.

```
1    Map<String,Object> map=new HashMap<String,Object>();
2    map.put("key","value");
3    map.put("number",Integer.valueOf(2));
4    assertEquals("value",StrLookup.mapLookup(map).lookup(
     "key"));
5    assertEquals("2",StrLookup.mapLookup(map).lookup("num
     ber"));
6    assertEquals(null,StrLookup.mapLookup(map).lookup("ot
     her"));
```

Example II. The extracted usage example by our approach.

Fig. 10. Instance analysis between Javadoc example (in the upper cell) and mined example (in the lower cell)

E. Threats to Validity

1) ***Construct Validity:*** Threats to construct validity are concerned with whether the data setup and measurement in the study reflects real-world situations. The main threat to construct validity is the way of building up the *GSD*. As we manually created the standard examples, errors and different opinions from participants would bias the evaluation result. To reduce this threat, we applied a three-round double-checked process in data setup and each test method in *GSD* is analyzed by at least two analysts.

2) ***Internal Validity:*** Threats to internal validity are concerned with the uncontrolled factors that may also be responsible for the evaluation results. In our experiment, the main threat to internal validity is the possible faults in the implementation of our approach. To reduce this threat, we exploited existing code analysis libraries, such as JDT AST. In addition, we reviewed all the code before conducting the experiment.

3) ***External Validity:*** Threats to external validity are concerned with whether the experimental results are generalizable for other situations. The main threat to external validity in our study lies in the representativeness of our subjects. To reduce this threat, we chose four subjects from different application domains. The complexity of test code also differs between projects.

VI. RELATED WORK

To make clear the obstacles to learning API, many studies [1], [2], [3] have investigated different factors and confirmed the significance of API usage examples. Robillard et al. analysed five categories of API documents and gave a classification of usage examples [1], [3].

There are many pieces of research on providing complementary API usage examples. One of the state-of-art approach compared in our experiment is *eXoaDoc*, which is proposed by Kim et al. [5], [6], [12]. Their approach consists of three modules: summarization module that search and summarize code examples, representation module that extracts features of candidate examples, organization module that clustering and ranking examples. They extracted code examples from client code repository and finally embedded the examples into the API documentation. Buse et al. [7] proposed an approach to automatically synthesizing human-readable usage examples by mining the abstract API usages.

These work employed code search engines, but there are some restrictions of the existing code search engines [12], [31], [32]. Thus much related work [8], [9], [10], [11] concentrated on searching for API usages more effectively and precisely from code repositories. They proposed their own code search engines or improved the existing engines to refine the search results. Some researchers integrated their engines with API documents [8], developing IDEs [10], [11] to assist the developers in finding the desired API examples. Thummalapenta et al. [10] mined method invocation sequences utilizing API's client code from large scale open source projects. Holmes et al. [16] used the code in local workspace as the repository of sample code.

Some other approaches (e.g., APIExample [13], Assieme [14], and Mica [15]) mined API examples from web pages. Wang et al. [13] collected API's related web pages from the web, and then analyzed the page contents to automatically identify and extract sample code snippets from pages.

API usage specification is another important kind of API learning resource. Zhong et al. [22] and Wu et al. [23] mined API specifications from different types of API data, such as code, comments, documentation, etc.

Nasehi et al. [17] investigated the feasibility of using test code as API usage examples. Their work provided us much inspiration. But they did not present an approach to address the problem. In our previous work [18], we proposed the idea of separating test scenarios according to code patterns. But the patterns are not general and the applicability is relatively low. Besides, the candidate examples are not further processed.

We adapted some techniques in our approach. Nguyen et al. [19] proposed a graph-based approach for mining usage patterns. Van Rompaey and some other researchers [28], [29] proposed and evaluated many strategies in establishing traceability links between unit tests and classes under test. As for clustering similar code examples, much research on detecting code clone or code similarity in repositories has been proposed [24], [25], [26].

VII. CONCLUSION

Existing approaches to extracting API usage examples mainly rely on client code or web sources, which are rarely available for new API libraries. In this paper, we have proposed a novel approach to mining API usage examples from test code. We have summarized a set of code patterns that characterizes the test methods containing multiple test scenarios. A heuristic code slicing approach based on the code patterns has been proposed to extract code examples from test methods. Our approach combines the technique of clustering to improve the representativeness of extracted examples. In an experimental evaluation on four open source projects, our approach performs much better than the *eXoaDoc* approach. A prototype tool, *UsETeC*, has been implemented for recommending usage examples from test code. We have applied the tool to open source projects and published the related resources online.

ACKNOWLEDGMENT

This research is sponsored by the National Basic Research Program of China (973) under Grant No. 2011CB302604, the High-Tech Research and Development Program of China (863) under Grant No. 2013AA01A605, the National Natural Science Foundation of China under Grant No. 61121063, U1201252, 61103024.

REFERENCES

[1] Robillard, M. P. What makes APIs hard to learn? Answers from developers. Software, IEEE, 26(6) (Nov. 2009), pp. 27-34.

[2] Scaffidi, C. Why are APIs difficult to learn and use?. Crossroads, 12(4) (May 2006), pp. 4-9.

[3] Robillard, M. P., and Deline, R. A field study of API learning obstacles. Empirical Software Engineering, 16(6) (Dec. 2011), pp. 703-732.

[4] Nasehi, S., Sillito, J., Maurer, F., and Burns, C. What makes a good code example?: A study of programming Q&A in StackOverflow. In *Proc. of ICSM'12.* (pp. 25-34).

[5] Kim, J., Lee, S., Hwang, S. W., and Kim, S. Adding examples into java documents. In *Proc. of ASE'09*. pp. 540-544

[6] Kim, J., Lee, S., Hwang, S. W., and Kim, S. Enriching Documents with Examples: A Corpus Mining Approach. *ACM Transactions on Information Systems (TOIS)*, 31(1) (2013), pp. 1.

[7] Buse, R. P., and Weimer, W. 2012. Synthesizing API usage examples. In *Proc. of ICSE'12*. pp. 782-792.

[8] Mar, L. W., Wu, Y. C., and Jiau, H. C. Recommending proper API code examples for documentation purpose. In *Proc. of APSEC'11*. pp. 331-338.

[9] Zhong, H., Xie, T., Zhang, L., Pei, J., and Mei, H. MAPO: Mining and recommending API usage patterns. In *Proc. of ECOOP'09.* pp. 318-343.

[10] Thummalapenta, S., and Xie, T. Parseweb: a programmer assistant for reusing open source code on the web. In *Proc. of ASE'07*. pp. 204-213.

[11] Sahavechaphan, N., and Claypool, K. October. XSnippet: mining for sample code. In *ACM Sigplan Notices* 41(10) (2006), pp. 413-430.

[12] Kim, J., Lee, S., Hwang, S. W., and Kim, S. May. Towards an Intelligent Code Search Engine. In *Proc. of AAAI'*10. pp. 1358-1363.

[13] Wang, L., Fang, L., Wang, L., Li, G., Xie, B., and Yang, F. APIExample: An effective web search based usage example recommendation system for Java APIs. In *Proc. of ASE'11*. pp. 592-595.

[14] Hoffmann, R., Fogarty, J., and Weld, D. S. October. Assieme: finding and leveraging implicit references in a web search interface for programmers. In *Proc. of UIST'07*. ACM. pp. 13-22.

[15] Stylos, J., and Myers, B. A. September. Mica: A web-search tool for finding API components and examples. In *Proc. of VL/HCC'06*. pp. 195-202.

[16] Holmes, R., and Murphy, G. C. May. Using structural context to recommend source code examples. In *Proc. of ICSE'05*. pp. 117-125.

[17] Nasehi, S. M., and Maurer, F. Unit tests as API usage examples. In *Proc. of ICSM'10*. pp. 1-10.

[18] Zhu, Z., Zou, Y., Jin, Y., and Xie, B. Generating API-usage example for project developers. In *Proc. of the 5th Asia-Pacific Symposium on Internetware*. 2013. pp. 34-37.

[19] Nguyen, T. T., Nguyen, H. A., Pham, N. H., Al-Kofahi, J. M., and Nguyen, T. N. Graph-based mining of multiple object usage patterns. In *Proc. of ESEC/FSE'09*. pp. 383-392.

[20] Wang, L., Zou, Y., Fang, L., Xie, B., and Yang, F. An Exploratory Study of API Usage Examples on the Web. In *Proc. of APSEC'12*. pp. 296-405.

[21] Liu, C., Zou, Y., Cai, S., Xie, B., and Mei, H. Finding the Merits and Drawbacks of Software Resources from Comments. In *Proc. of ASE'11*. pp.432-435.

[22] Zhong, H., Zhang, L., Xie, T., and Mei, H. Inferring Resource Specifications from Natural Language API Documentation. In *Proc. of ASE'09*. pp. 307-318.

[23] Wu, Q., Liang, G. T., Wang, Q. X., and Mei, H. Mining effective temporal specifications from heterogeneous API data. *Journal of Computer Science and Technology*, 26(6), pp. 1061-1075.

[24] Huang, L., Shi, S., and Huang, H. 2010. A new method for code similarity detection. In *Proc. of PIC'10*. Vol. 2, pp. 1015-1018.

[25] Baxter, I. D., Yahin, A., Moura, L., Sant'Anna, M., and Bier, L. Clone detection using abstract syntax trees. In *Proc. of ICSM'98*. pp. 368-377.

[26] Wise, M. J., Detection of similarities in student programs: YAP'ing may be preferable to plague'ing. In *ACM SIGCSE Bulletin*. 24(1) (Mar. 1992). pp. 268-271.

[27] Li, B. X., Fan, X. C., Pang, J., and Zhao, J. J. A model for slicing Java programs hierarchically. *Journal of Computer Science and Technology*, 19(6), pp. 848-858.

[28] Van Rompaey, B., and Demeyer, S. Establishing traceability links between unit test cases and units under test. In *Proc. of CSMR'09*. pp. 209-218.

[29] Qusef, A., Oliveto, R., and De Lucia, A. Recovering traceability links between unit tests and classes under test: An improved method. In *Proc. of ICSM'10*. pp. 1-10.

[30] Levenshtein Distance, http://en.wikipedia.org/wiki/Levenshtein_distance.

[31] Bajracharya, S., Ossher, J., and Lopes, C. Searching API usage examples in code repositories with sourcerer API search. In *Proc. of SUITE'10*. pp. 5-8.

[32] Ossher, J., Bajracharya, S., Linstead, E., Baldi, P., and Lopes, C. Sourcererdb: An aggregated repository of statically analyzed and cross-linked open source java projects. In *Proc. of MSR'09*. pp. 183-186.