2011 18th Asia-Pacific Software Engineering Conference

Recommending Proper API Code Examples for Documentation Purpose

Lee Wei Mar*, Ye-Chi Wu*, Hewijin Christine Jiau†
*Institute of Computer and Communication Engineering
†Department of Electrical Engineering
National Cheng Kung University, Taiwan
{lwmar, yechiwu}@nature.ee.ncku.edu.tw, jiauhjc@mail.ncku.edu.tw

***Abstract*—Code examples are important resources for expressing correct application programming interface (API) usages. However, many framework and library APIs fail in offering sufficient code examples in corresponding API documentations. This is because constructing proper code examples for documentation purpose takes significant developers' efforts. To reduce such effort, this work proposes a methodology, PropER-Doc, that recommends proper code examples for documentation purpose. PropER-Doc accepts queries from API developers and utilizes code search engines (CSEs) to collect corresponding code example candidates. The structural and conceptual links between API elements are captured from the API implementation and available API documents to guide candidate recommendation. During recommendation, PropER-Doc groups collected candidates based on involved API types for distinguishing different API usages. To assist API developers in selecting proper candidates, a diagrammatic presentation and three code example appropriateness metrics are also developed in PropER-Doc. Two case studies on Eclipse JDT framework are conducted to confirm the effectiveness of PropER-Doc.**

***Keywords*-framework; documentation; API usage; code example;**

I. Introduction

During object-oriented software applications development, programming with existing Application Programming Interfaces (APIs) becomes a common practice. As existing object-oriented frameworks become more and more powerful, the complexity of offered APIs increased accordingly. Such complexity put barriers for correctly using of APIs [1]. To guide programmers to use a framework API correctly, API developers provide an API documentation as the programming reference. The API documentation is an important resource for programmers while learning and using an API [14]. Therefore, making an API documentation effective for guiding programmers using API correctly is a critical job for API developers.

One feasible method for ensuring the effectiveness of API documentation is to augment documentation with credible code examples [13], [14]. With the credible code examples, programmers can learn correct API usage by imitation [8], [15]. In practice, some frameworks, such as SWT GUI framework [19], do provide API documentation with rich set of credible code examples. Nevertheless, most existing APIs still fail to provide sufficient code examples in corresponding documentation [7]. To construct code examples for documentation purpose, huge human effort is required for confirming correct API usages and constructing corresponding code examples from scratch. Therefore, there is an urgent need to reduce API developers' effort in such process.

To eliminate developers' effort on constructing proper code examples for documentation purpose, Kim et al. [7] developed a technique that automatically extracts code examples from API client code and adds them to API documents. However, applying the technique, as a trade off of automation, developers are not involved in checking the extracted code examples. Consequently, improper code examples may be added to API documents and contradictorily downgrade the correctness of API documents. Furthermore, this technique reports code examples that demonstrate the use of single API method. While these reported code examples are good references for fine-grained API call adaptation, they do not explain the whole object interaction scenarios for assisting programmers in learning typical API usages.

In this work, a methodology named PropER-Doc (**Prop**er **E**xample **R**ecommendation for **Doc**umentation), is proposed. Considering that an API type is a basic manipulation unit during using API, PropER-Doc aims to recommend code examples that demonstrate whole object interaction scenario during using a specific API type. Given an API type, PropER-Doc collects code example candidates from code search engines (CSEs) [10], [2] and recommends candidates that meet correct API type usages to developers. To guarantee the effectiveness of the recommendation, PropER-Doc constructs structural and conceptual links based on the API implementation and available API documents. These links capture implicit API element usage dependencies and are served as recommendation bases in PropER-Doc. In addition, to incorporate with developers in navigating and selecting candidates, three recommendation assistants are offered in PropER-Doc:

1) a candidate grouping algorithm that groups candidates for distinguishing different API usages;
2) a diagrammatic presentation technique that summarizes object interaction of a specific API type usage;
3) three code example appropriateness metrics for assessing the quality of a specific candidate.

1530-1362/11 $26.00 © 2011 IEEE
DOI 10.1109/APSEC.2011.18

```
char[] source = ...; // source code as char[]
ASTParser p = ASTParser.newParser(AST.JSL3);
p.setKind(ASTParser.K_COMPILATION_UNIT);
p.setSource(source);                                          (a)
p.setProject(aJavaProject);
p.setUnitName(javaClassName);
p.setResolveBinding(true);
CompilationUnit n = (CompilationUnit) p.createAST(monitor);
```

```
ICompilationUnit unit = ...; // an ICompilationUnit object
ASTParser p = ASTParser.newParser(AST.JSL3);
p.setKind(ASTParser.K_COMPILATION_UNIT);
p.setSource(unit);                                            (b)
p.setResolveBinding(true);
CompilationUnit n = (CompilationUnit) p.createAST(monitor);
```

```
ICompilationUnit[] units = ...; // an ICompilationUnits array
ASTParser parser = ASTParser.newParser(AST.JSL3);
p.setKind(ASTParser.K_COMPILATION_UNIT);
p.setProject(aJavaProject);                                   (c)
p.setResolveBinding(true);
p.createASTs(units, bindingKeys, astRequestor, monitor);
```

Figure 1. Three representative `ASTParser` usage code examples. API usages differences are marked in bold. (a) get an AST from source code of a Java class, (b) get an AST from an `ICompilationUnit` object, (c) get ASTs from a list of `ICompilationUnit` objects.

II. Challenges of Proper Code Example Construction

An example is used to explain the challenges that API developers face during constructing code examples for documentation purpose. In the example, the class `ASTParser` (`org.eclipse.jdt.core.dom.ASTParser`) in Eclipse JDT framework [18] is selected as the subject. `ASTParser` is a Java language parser. It accepts different types of objects as transformation sources for creating corresponding abstract syntax trees (ASTs) with binding information. Figure 1 lists three representative `ASTParser` code examples for different transformation sources. Given different transformation sources, `ASTParser` needs to be configured in different ways for achieving the transformation. Such a complex configuration style makes the use of `ASTParser` difficult for inexperience programmers. To guide programmers to use `ASTParser` properly, clear explanation on its different usages is necessary in API documentation. However, official `ASTParser` Javadoc document[2] only offers one 4-line code example to show the basic usage flow. Furthermore, the explanation of `ASTParser` configuration is scattered over different parts of the document. To understand correct `ASTParser` usages with available document, significant human cognitive effort is required for programmers to traverse through the document. Since the code examples in Figure 1 well explain correct `ASTParser` usages, adding them to the document is appropriate.

From our observation, to cover different `ASTParser` usages, nine representative code examples are required. To construct representative code examples for documentation purpose, an API developer must identify all important API usages that need code examples by himself. Then, for each API usage, the developer has to build corresponding code example from scratch. Such a manual code examples construction process is laborious for the developer. Instead, finding important API usages and collect proper code examples from available API client code seems to be a promising approach for effort reduction. Here, Google code search[3] is utilized to find proper API client code that demonstrates the use of `ASTParser` class. Keyword such as the full name of `ASTParser` is taken as input of Google code search, which then returns a list of source file entries that match the keyword as search results. In fact, the search results indeed contain promising code example candidates that cover all correct `ASTParser` usages. However, there are several issues needed to be resolved for making such approach practical.

- **Locating code example candidates.**
  A large portion of CSEs search results are false positives. For instance, among 1873 methods defined in the top 100 `ASTParser` search results, only 6% (114/1873) of them invoke API methods defined in `ASTParser`. The developer faces difficulty in locating these relatively small set of candidates from search results.
- **Reorganizing code example candidates.**
  Once candidates are located, the developer needs proper organization of candidates to explore them in a higher level of abstraction. Without proper organization, the developer has to inspect each candidate in detail locally. As a result, the developer has difficulty in systematically distinguishing important API usages demonstrated by the collected candidates.
- **Checking code example candidates appropriateness.**
  In selecting proper code example candidates, developers must check the appropriateness of a candidate on explaining a correct API usage. During checking, the developer has to consider many factors, such as the completeness of the candidate, the portion of irrelevant code in the candidate and so on. Checking a candidate by manually code inspection is time-consuming and laborious for the developer.

III. A Methodology for Proper Code Examples Recommendation

Aiming at resolving identified issues during finding proper code examples, a recommendation methodology, PropER-Doc (**Prop**er **E**xample **R**ecommendation for **Doc**umentation), is proposed. In PropER-Doc, API element links are constructed based on the API implementation and available API documents to model implicit API element usage dependencies. PropER-Doc takes these API element links as bases for recommendation to keep the suggested candidates appropriate for documentation.

Figure 2 shows the process overview of PropER-Doc. At beginning, an API developer specifies the name of an API

[2]http://help.eclipse.org/helios/index.jsp?topic=/org.eclipse.jdt.doc.isv/reference/api/org/eclipse/jdt/core/dom/ASTParser.html

[3]http://www.google.com/codesearch

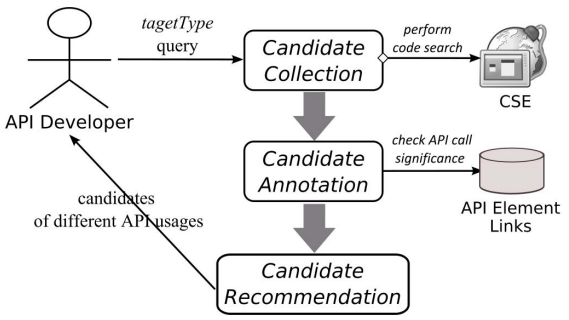

Figure 2. Process overview of PropER-Doc.

Table I
DIFFERENT KINDS OF LINKS BETWEEN API ELEMENTS. THE NOTATION T, I, E REPRESENTS API TYPE, API INVOCATION UNIT (METHOD AND FIELD), AND API ELEMENT, RESPECTIVELY. E = {T, I}.

Link	Type	Description
def(t, i)	T × I	t defines i as a method or field
ext(t_1, t_2)	T × T	t_1 is a descendent of t_2
use(i, t)	I × T	signature of i refers to t
con(e_1, e_2)	E × E	document content of e_1 refers to e_2

type that needs code examples (referred as *targetType*) as a query. Then, the code example candidates of *targetType* are *collected* from CSEs. These candidates are *annotated* based on API element links constructed in advance. Finally, candidates of different API usages are *recommended* to the API developer.

A. Constructed Links between API Elements

The links between API elements are constructed to indicate API elements expected to be used together from API developers' viewpoint. A link between two API elements is defined as a tuple (x, y), where x and y are structurally or conceptually relevant API elements. Table I lists four different kinds of links between two API elements.

In Table I, the top three kinds of links are the structural links. Structural links are API element structural relations maintained in the API implementation, such as type hierarchy and containment relationship. The last one kind of link is the conceptual link. Conceptual links are constructed based on the descriptive content of available API documents. If the document description of an API element x refers to another API element y, then a conceptual link from x to y is created. These conceptual links capture structurally hidden but important relations between API elements.

B. Candidate Collection and Annotation

PropER-Doc delegates the code search responsibility to CSEs for collecting potential source files with code examples. For a long lived API with rich set of client code on web, web-based CSEs such as Google code search and Koder[4] can

[4]http://www.koders.com/

Table II
SIGNIFICANT LEVELS FOR ANNOTATING API CALLS IN A CANDIDATE *C*.

Lv.	Formulation
6	{call | ∀ call ∈ *C*, def(*targetType*, call)}
5	{call | ∀ call ∈ *C*, con(*targetType*, t) ∧ def(t, call)}
4	{call | ∀ call ∈ *C*, use(call, *targetType*)}
3	{call | ∀ call ∈ *C*, def(*targetType*, i) ∧ (con(i, call) ∨ (con(i, t) ∧ def(t, call)))}
2	{call | ∀ call ∈ *C*, AccessingType(call) = *targetType* ∧ ext(*targetType*, t) ∧ def(t, call)}
1	{call | ∀ call ∈ *C*, *targetType* ∈ ParameterTypes(call) ∧ ext(*targetType*, t) ∧ use(t, call) }

be directly utilized for code search. On the contrary, if the API is used within private organizations or it is in the early stage of promotion, API developers can construct a dedicated CSE with the available code search infrastructure [10] for efficient client code collection, management and search.

Once a *targetType* is accepted as a query, it is delegated to a CSE (or several CSEs). The CSE searches in the available open source software on the web and return relevant source files. PropER-Doc parses each source file and checks the invoked API calls inside. For each method implementation of a source file, if it invokes API calls defined within *targetType*, then that method implementation is extracted as a code example candidate.

After collecting candidates from the CSE, important API calls for demonstrating *targetType* within each candidate are captured and annotated based on API element links. At first, all API calls in a candidate are extracted. For each API call, its relevance to *targetType* will be evaluated. If the API call is considered relevant to *targetType*, it is annotated with a level as an indicator of its importance on demonstrating *targetType* usage. The higher the level, the more important the API call.

Table II lists the formulations for different levels. An API call defined in *targetType* is considered as most important one and is annotated with level 6. If an API call is defined in an API type t which is referred by document content of *targetType*, it is annotated with level 5. Such API call gains higher level because t is important and intended to be used together with *targetType*. If the signature of an API call takes *targetType* as parameter type or return type, it is annotated with level 4 because of the structural dependency. For an API call referred by document content of API elements, it is annotated with level 3. An inherited API call invoked on an object of *targetType* is annotated with level 2. Finally, an API call is annotated with level 1 while it takes an object of *targetType* as parameter by upcasting.

C. Candidate Recommendation

According to annotated API calls in candidates, the code example candidates are reorganized and presented to the de-

veloper. At first, candidates are divided into different groups to assist the developer in checking different *targetType* usages. Then, for each group, a diagrammatic representation of the group is presented to give the developer an overview on the object interaction of corresponding usage. Once the developer decides to pick code example from a group, candidates are ranked based on three appropriateness metrics and listed to the developer.

```
   input : an API type targetType
   input : a code example candidate C
   output: a set iTypes
 1 Let iTypes be an empty set;
 2 add targetType to iTypes;
 3 foreach call ∈ AnnotatedCalls(C) do
 4     if InvokingType(call) == targetType then
 5         add ParameterTypes(call) to iTypes;
 6     end
 7     else if ReturnType(call) == targetType then
 8         add InvokingType(call) to iTypes;
 9     end
10     else if targetType ∈ ParameterTypes(call) then
11         add InvokingType(call) to iTypes;
12     end
13 end
14 return iTypes;
```

Algorithm 1: Get *iTypes* of *targetType* in candidate *C*.

1) Grouping Candidates based on Interacted API Types: Different API usages usually involve interactions among objects of different API types. To distinguish candidates of different API usages, a candidate grouping algorithm is developed. The algorithm group candidates based on *iTypes*, the set of API types that directly interact with *targetType* inside the candidate. Algorithm 1 shows the procedure that identifies *iTypes* of a candidate *C*. For each annotated API call in *C*, if the invoking object type of the API call *targetType*, then the object types of parameters in that API call are added to *iTypes*. If the API call refers *targetType* in return type or parameters types, the invoking object type of the API call is invoked is also added to *iTypes*.

Algorithm 2 illustrates the grouping algorithm for different interacted API types. The algorithm consists of two stages. In the first stage (Algorithm 2 line 2-10), candidates with the same *iTypes* are put into a same group. In the second stage (Algorithm 2 line 11-26), group merging procedure is performed based on the inclusion relationship. Considering two groups g_i and g_j, where the set of annotated API calls in g_i is a subset of those in g_j. In such case, the API usages demonstrated by g_i are also covered by g_j. So, g_i is merged to g_j for getting more compact grouping result.

With the grouping result, the developer can systematically explore candidates of different API usages in an abstraction of object interaction level. Figure 3 (a) lists the grouping result of `ASTParser` code example candidates from top 100 source files returned by Google code search. The developer

```
   input : an API type targetType
   input : a set of code example candidate Cs
   output: a dictionary Gd
 1 Let Gd be an empty dictionary;
 2 foreach c ∈ Cs do
 3     Let key = iTypes(targetType, c);
 4     Let group be an empty list;
 5     if Gd.hasKey(key) then
 6         group = Gd.value(key);
 7     end
 8     add c to group;
 9     Gd.put(key, group);
10 end
11 Let Done = False;
12 while not Done do
13     Done = True;
14     Let t_i, t_j, g_i, g_j be null;
15     Let KeyPairs = Gd.keys() × Gd.keys();
16     foreach (k_i, k_j) ∈ KeyPairs do
17         g_i = Gd.value(k_i), g_j = Gd.value(k_j);
18         if g_i ≠ g_j and ⋃_{c_i∈g_i} AnnotatedCalls(c_i)
           ⊆ ⋃_{c_j∈g_j} AnnotatedCalls(c_j) then
19             add g_i to g_j;
20             Gd.remove(k_i);
21             Gd.put(k_j, g_j);
22             Done = False;
23             break;
24         end
25     end
26 end
27 return Gd;
```

Algorithm 2: A candidate grouping algorithm for different interacted API types.

can select one group for further checking candidates in that group.

2) API Type Interaction Diagram: Once the developer selects a group, an API type interaction diagram is presented to help the developer check the summary of candidates in the group. Figure 3 (b) shows the API type interaction diagram of Figure 3 (a) group 1. In the diagram, API types are presented with boxes. The edges between boxes represent the API invocation relationships of these candidates. The API calls that contribute to a specific invocation relationship are listed by the corresponding edge with the invocation popularity. Unpopular API calls will be hidden at first for reducing the visual complexity of the diagram. With the API type interaction diagram, the developer gets an overview about API type interaction of candidates in the group. This diagram is also useful in helping the developer pick candidates of interest. The developer can choose interested API calls in the diagram, and then the candidates that contain all interested API calls in the group would be presented. For instance, Figure 3 (b) indicates that candidates in group 1 often use `newParser()`, `setSource()`, `setResolveBinding()`, and `createAST()` methods of `ASTParser` during parsing an `ICompilationUnit` object.

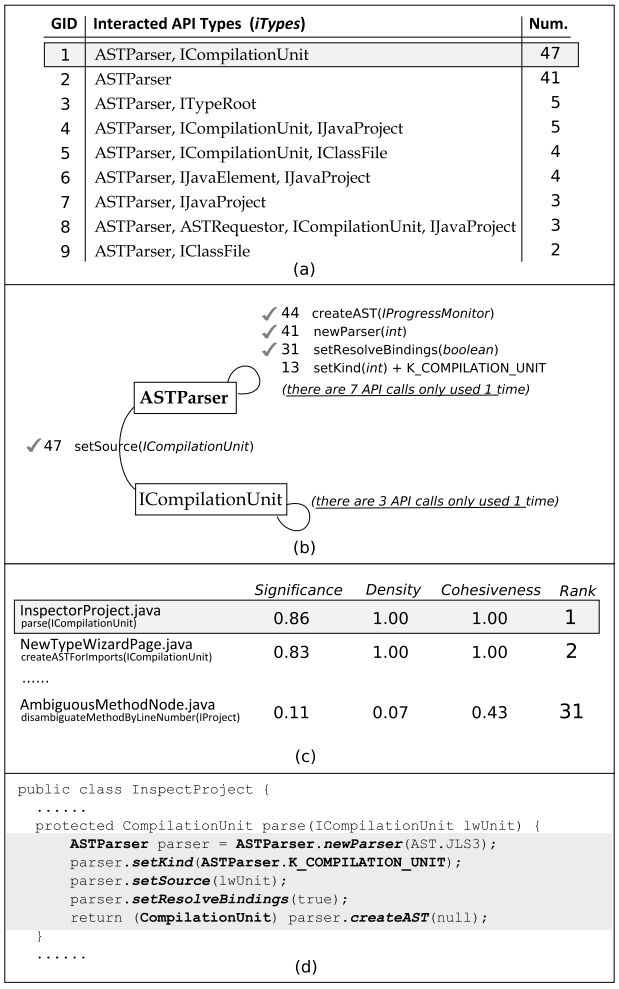

Figure 3. Code example candidates recommendation. (a) grouping result based on interacted API types, (b) API type interaction diagram of group 1, (c) selected candidates with appropriateness metrics ranking, (d) code presentation for a given candidate.

If the developer chooses those frequently used API calls in Figure 3 (b), corresponding candidates that invokes all those API calls would be listed (Figure 3 (c)).

3) Ranking based on Appropriateness Metrics: PropER-Doc performs ranking during presenting the candidate list to assist developer in picking candidates. Three appropriateness metrics: significance, density, and cohesiveness, are defined to evaluate the quality of a candidate for ranking. The formulation of appropriateness metrics is based on the annotated API calls in candidates. The range of these metrics are between 0 to 1. All the metrics are positive correlated with the quality of a candidate. The higher the metric value, the more appropriate the candidate for documentation purpose.

- **Significance.** The significance metric evaluates the importance of all referred API calls in a candidate with respected to correct *targetType* usages. A candidate with higher significance is tended to be more representative for demonstrating a specific *targetType* usage. The significance metric for a candidate C is formulated using annotated levels of referred API calls.

$$significance(C) = \frac{\sum_{call \in C} Level(call)}{6 \times CallNum(C)}$$

- **Density.** The density metric evaluates the portion of code lines that refers annotated API calls in a candidate. A candidate with higher density is tended to be more comprehensible. This is because the effort required in filtering out noisy information during code inspection can be reduced.

$$density(C) = \frac{AnnotatedLineNum(C)}{LineNum(C)},$$

- **Cohesiveness.** The cohesiveness metric models aggregation level of annotated API calls within a candidate. For a candidate with higher cohesiveness, the developer can extract code example from it with less effort. In computing cohesiveness metric of a candidate C, the API dense blocks are identified from C. An API dense block is a code block in C and every code line in that block contains annotated API calls. A block weight function is defined to evaluate the aggregation degree of annotated API calls in a block. The weight of a block is proportional to the line number of the block and the number of annotated API calls inside it.

$$Weight(b) = LineNum(b) \times AnnotatedCallNum(b)$$

By considering the distribution of API dense blocks, the cohesiveness metric of C is defined as the ratio between total weight of relevant blocks and weight of a compact block, B, that contains all annotated API calls in C.

$$Cohesiveness(C) = \frac{\sum_{b \in APIDenseBlocks(C)} Weight(b)}{Weight(B)}$$

The candidate list is ranked according to the summation of appropriated metrics, as shown in Figure 3 (c). The developer can select one specific candidate in Figure 3 (c) to check its source code. In source code presentation (Figure 3 (d)), the code lines with annotated API calls in the candidate are highlighted for assisting inspection. The developer can identify relevant code, perform minor modifications, and then extract needed code block from the candidate for documentation.

D. Implementation

PropER-Doc has been realized using Java language. Currently, PropER-Doc implementation supports the recommendation of code examples for a framework API implemented in Java. Google code search is used as the default CSE for performing the code search. Besides Google code search, PropER-Doc implementation can be configured to work together with additional CSEs when needed.

During constructing structural API element links, the Java

Table III
MAPPING BETWEEN CORRECT API USAGES AND GROUPING RESULT: ASTPARSER CASE.

GID	Num.	Mapped API Usage	Effort	
			ΔStmt(+/−)	Coh.
1	47	Get an AST from an `ICompilationUnit` object	0 / 0	1.00
2	41	Get an AST subtree from a string of (a) a class body	0 / 0	1.00
		(b) a list of statements	0 / 0	1.00
		(c) an expression	4 / 0	1.00
3	5	Get an AST from an `ITypeRoot` object	0 / 1	1.00
4	5	– (Similar to GID 1 with redundant configuration)	–	–
5	4	– (Mixed API usages)	–	–
6	4	Get binding information of given `IJavaElements`	3 / 0	1.00
7	3	Get an AST from source code of a Java class	0 / 0	1.00
8	3	Get an ASTs from given `ICompilationUnits`	16 / 0	0.63
9	2	Get an AST from an `IClassFile` object	0 / 0	1.00

AST parser offered in Eclipse JDT framework is utilized to perform code analysis on the API implementation. The JDT AST parser is also used in parsing collected candidates. During parsing a candidate, the AST parser provides the binding information for the program elements in the candidate. By checking the binding information, the referred API calls in the candidate are identified for annotation.

To construct conceptual API element links based on Javadoc-based API documents [9], PropER-Doc performs HTML parsing on the descriptive content of documents. Given the descriptive content for an API element, if it refers to another API element as a hyper-link or in a HTML `<code>` tag, then a conceptual link is created between these two elements. The jericho[5] HTML parser is used to perform parsing on Javadoc web pages.

IV. EVALUATION

Two case studies are performed to evaluate the recommendation effectiveness of PropER-Doc. Two commonly used complex API types in Eclipse JDT framework, `ASTParser` (`org.eclipse.jdt.core.dom.ASTParser`) and `SearchEngine` (`org.eclipse.jdt.core.search.SearchEngine`), are chosen as study subjects. In the studies, PropER-Doc accepts top 100 source files returned from the Google code search during candidate collection.

A. *ASTParser Case*

There are nine different `ASTParser` API usages that need representative code examples. Five of them accept different objects for constructing an AST with binding information. Three of them accept different constructs of Java source code for constructing an AST subtree. The last one accepts a Java program element and create the binding information of that element. All these `ASTParser` API usages differ from each other in the configuration way or in the returned processing result.

In `ASTParser` case, PropER-Doc gives 114 extracted candidates and splits them into nine groups. The group id (GID), interacted API types in candidates of each groups, and the amount of candidates has been summarized in Figure 3 (a). For further understanding mapping between candidates of each group and corresponding API usages, the candidates of each group are inspected with the help of the API type interaction diagrams. The inspection result is summarized in Table III.

In recommending code example candidates for `ASTParser`, there are seven groups covering all nine correct `ASTParser` usages. Six of these seven groups have one to one mapping to `ASTParser` usages. This indicates that the grouping algorithm is effective in reorganizing `ASTParser` candidates for distinguishing different API usages. Group 2 covers three different `ASTParser` usages for generating ASTs using different kind of constructs to be parsed from Java source code. In this situation, API type interaction diagram can further assist developers in selecting candidates that demonstrate specific API usage. The three usages utilize different constant parameters during setting the `setKind()` method. Therefore, developers can select different constant parameters for `setKind()` method in the API type interaction diagram, then candidates that match each of these three usages are quickly identified.

We further check the effort needed for transforming candidates to representative code examples. For each `ASTParser` usage, a top ranked candidate in overall ranking result is selected for inspection. For group 2, the API type interaction diagram is used for selecting candidates of distinct `ASTParser` usages. The selected candidates are inspected for checking the effort required to transform those candidates to representative code examples. The inspection result is summarized in **Effort** column of Table III[6]. In the column, the **ΔStmt** sub-column records the number of redundant (+) or missed (−) statements of candidates comparing to representative code examples. The **Coh.** sub-column records the cohesiveness metric of candidates.

Among nine selected candidates, five of them (group 1, 2(a), 2(b), 7, and 9) perfectly match representative code examples for corresponding `ASTParser` usages. Three can-

[5]http://jericho.htmlparser.net/docs/index.html

[6]The inspected candidates are available on http://colors.ee.ncku.edu.tw/ProperDoc/CaseStudy

Table IV
MAPPING BETWEEN CORRECT API USAGES AND GROUPING RESULT: SEARCHENGINE CASE.

GID	Num.	Mapped API Usage	Effort	
			ΔStmt(+/−)	Coh.
1	5	*mechanism*$_{enclose}$	0 / 0	1.00
2	5	*mechanism*$_{type}$	0 / 1	1.00
3	1		0 / 1	1.00
4	1		4 / 0	1.00
5	1		10 / 0	1.00
6	45	*mechanism*$_{general}$	0 / 1	1.00
7	19		0 / 0	1.00
8	2		0 / 2	1.00
9	30	– (Non-representative API usage)	–	–
10	5		–	–

didates (group 2(c), 6, and 8) covers needed API calls for corresponding usages, and all of them have high cohesiveness value (from 0.63 to 1.00). This indicates that the needed API calls within these three candidates are tightly aggregated and can be easily extracted as code examples. For the top ranked candidate in group 3, only one API call is missed, and the effort for modification is relatively minor. In summary, ranking candidates according to appropriated metrics greatly assists developers in choosing proper candidates.

B. `SearchEngine` Case

The `SearchEngine` class in JDT provides support to search Java program elements in specific scope that matches given specification. `SearchEngine` offers one general search mechanism and two different handy search mechanisms. In performing a general search mechanism (*mechanism*$_{general}$), `SearchEngine` requires a search pattern, a list of search participant, a search scope, and a requestor object. The first handy search mechanism (*mechanism*$_{enclose}$) performs search for Java program elements enclosed in specified Java program element. The second handy mechanism (*mechanism*$_{type}$) performs search for Java types that match the given string in the given search scope. Different code examples are required to demonstrate the use of different search mechanisms.

PropER-Doc returns 114 candidates from top 100 source files and splits them into ten groups. By inspecting API type-based diagram of each group, the mapping between groups and different `SearchEngine` search mechanisms is summarized in Table IV. There are eight valid groups that correctly demonstrate `SearchEngine` usages. Candidates in group 1 demonstrate *mechanism*$_{enclose}$. Candidates from group 2 to 5 demonstrate *mechanism*$_{type}$ with different requestor objects and search scopes. Candidates from group 6 to 8 demonstrate *mechanism*$_{general}$ with different search scopes.

In Table IV, *mechanism*$_{type}$ and *mechanism*$_{general}$ are mapped to multiple groups. Such many to one mapping indicates that these two search mechanisms may involves different combination of interacted API types. This reminds developers that `SearchEngine` offers highly flexible API calls in performing these two search mechanisms. Therefore, more than one representative code examples are needed for demonstrating the flexibility of these mechanisms. The multiple groups mapped to these mechanisms can assist developers in selecting various representative code examples.

In `SearchEngine` case, the effort needed for transforming top ranked candidates to representative code examples is also checked. The result is summarized in **Effort** column of Table IV. In these eight valid groups, the top ranked candidates of two groups perfectly match the correct `SearchEngine` API usages. The top ranked candidates of group 4 and 5 covered all needed API calls, and these API calls are tightly aggregated (cohesiveness value = 1.00) and can be easily extracted. The top ranked candidates of group 2, 3, 6 and 8 misses one to two needed API calls. For group 2 and 6, the missed API call can be complemented by checking other candidates of the same groups. For group 3 and 8, which have merely one or two candidates inside, developers can refer to similar groups that mapped to the same usage for picking up the missed API calls.

The study results confirm that PropER-Doc is effective for assisting developers in identifying different API usages. Also, PropER-Doc greatly help developers pick good candidates that can be transformed to representative code examples with less modification effort.

V. RELATED WORK

Kim et al. [6], [7] proposed a technique that automatically augments API documents with code examples. In their work, code examples are selected to demonstrate the use of single API method. The technique performs clustering and ranking based on referred API calls in candidates to select representative code examples. Since their work focuses on finding code examples for each API method, the code example appropriateness with respect to the correct API type usage is not taken into account in their work.

Several recommendation systems have been developed to suggest relevant code examples for supporting API programming tasks. Strathcona [5] recommends code examples relevant to the programming context where a programmer involves. Prospector [11] and XSnippet [16] accept an API type query pair (T_{in}, T_{out}) from a programmer, and suggest code examples that illustrate the uses of T_{in} object to obtain T_{out} object. PARSEWeb [20] also recommends code examples for API object instantiation using web-based CSEs. MAPO [22] applies data mining techniques on API client code for finding frequent API call sequences as API usage patterns. When programmers ask for the usage of a specific API call, MAPO would present relevant API usage patterns along with code examples as guidance. All the aforementioned systems focus on recommending code examples that directly match programmers needs. Alternatively, PropER-Doc aims to assist developers in selecting code examples for documentation purpose. Rather than directly recommending code examples, PropER-Doc mainly concerns the cooperation with developers for effective organization, navigation

and selection of collected candidates.

In our former work, CoDocent [21] is proposed to support example-based API learning with API documents. While inspecting a code example for API learning, CoDocent suggests API documents relevant to the API calls in the code example. These suggested API documents will help programmers gain a comprehensive understanding of the API usage demonstrated by the code example. As API documents contain comprehensive information about API usages, they have also been used to support different activities such as API code search and API specification mining. In API code search tool development, Mica [17] uses the content of API documents to refine the code search result. SNIFF [2] and Exemplar [3] further use API documents for code search query expansion. In API specification mining, Doc2Spec [23] utilizes nature language processing techniques to infer API specifications from API documents. Inspired from these works, PropER-Doc identifies conceptual links between API elements from API documents to guide proper code example recommendation. Also, PropER-Doc utilizes the techniques that measure API popularity [4], [12] to assist in representative example selection.

VI. Conclusion

In this work, a methodology, PropER-Doc, is proposed to recommend proper code examples for documentation purpose. PropER-Doc accepts an API type name as the query and utilizes CSEs to collect code example candidates. In PropER-Doc, a grouping algorithm is developed to distinguish candidates of different API usages. Also, an API type interaction diagram and three appropriateness metrics are proposed to assist recommended candidates navigation and selection. The effectiveness of PropER-Doc is evaluated through two case studies. The studies results confirm that, given an API type, PropER-Doc can distinguish its various usages and recommend credible candidates as documentation materials.

Acknowledgment

Part of this work has been supported by National Science Council, Taiwan (NSC 97-2221-E-006-177-MY3).

References

[1] J. Bosch, P. Molin, M. Mattsson, and P. Bengtsson, "Object-oriented framework-based software development: Problems and experiences," *ACM Comp. Survey*, vol. 32, no. 1, pp. 3–8, Mar. 2000.

[2] S. Chatterjee, S. Juvekar, and K. Sen, "SNIFF: a search engine for java using Free-Form queries," in *Int'l Conf. Fundamental Approaches to Softw. Eng.*, 2009, pp. 385–400.

[3] M. Grechanik, C. Fu, Q. Xie, C. McMillan, D. Poshyvanyk, and C. Cumby, "A search engine for finding highly relevant applications," in *Int'l Conf. Softw. Eng.*, 2010, pp. 475–484.

[4] R. Holmes and R. J. Walker, "Informing Eclipse API production and consumption," in *OOPSLA Wkshp. Eclipse Technology eXchange*, 2007, pp. 70–74.

[5] R. Holmes, R. J. Walker, and G. C. Murphy, "Approximate structural context matching: An approach to recommend relevant examples," *IEEE Trans. Softw. Eng.*, vol. 32, no. 12, pp. 952–970, Dec. 2006.

[6] J. Kim, S. Lee, S. won Hwang, and S. Kim, "Adding examples into java documents," in *Int'l Conf. Automated Softw. Eng.*, 2009, pp. 540–544.

[7] J. Kim, S. Lee, S. won Hwang, and S. Kim, "Towards an intelligent code search engine," in *Proc. 24th AAAI Conf. Artificial Intelligence*, 2010, pp. 1358–1363.

[8] A. J. Ko, B. A. Myers, and H. H. Aung, "Six learning barriers in end-user programming systems," in *IEEE Symp. Visual Lang. and Human-Centric Comp.*, 2004, pp. 199–206.

[9] D. Kramer, "API documentation from source code comments: a case study of javadoc," in *Int'l Conf. Computer Documentation*, 1999, pp. 147–153.

[10] E. Linstead, S. Bajracharya, T. Ngo, P. Rigor, C. Lopes, and P. Baldi, "Sourcerer: mining and searching internet-scale software repositories," *Data Mining and Knowledge Discovery*, vol. 18, no. 2, pp. 300–336, Apr. 2009.

[11] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman, "Jungloid mining: helping to navigate the API jungle," *ACM SIGPLAN Not.*, vol. 40, no. 6, pp. 48–61, Jun. 2005.

[12] Y. Mileva, V. Dallmeier, and A. Zeller, "Mining API popularity," in *Proc. 5th Int'l Academic and Industrial Conf. Testing - Practice and Research Techniques*, 2010, pp. 173–180.

[13] J. Nykaza, R. Messinger, F. Boehme, C. L. Norman, M. Mace, and M. Gordon, "What programmers really want: results of a needs assessment for SDK documentation," in *Int'l Conf. Computer Documentation*, 2002, pp. 133–141.

[14] M. P. Robillard, "What makes APIs hard to learn? answers from developers," *IEEE Softw.*, vol. 26, no. 6, pp. 27–34, Aug. 2009.

[15] M. B. Rosson and J. M. Carroll, "The reuse of uses in smalltalk programming," *ACM Trans. on Computer-Human Interaction*, vol. 3, no. 3, pp. 219–253, Sep. 1996.

[16] N. Sahavechaphan and K. Claypool, "XSnippet: mining for sample code," *ACM SIGPLAN Not.*, vol. 41, no. 10, pp. 413–430, Oct. 2006.

[17] J. Stylos and B. A. Myers, "Mica: A Web-Search tool for finding API components and examples," in *Proc. Visual Lang. and Human-Centric Comp.*, 2006, pp. 195–202.

[18] The Eclipse Foundation, "Eclipse java development tools (JDT)," http://www.eclipse.org/jdt/, 2011.

[19] The Eclipse Foundation, "SWT: The standard widget toolkit," http://www.eclipse.org/swt/, 2011.

[20] S. Thummalapenta and T. Xie, "PARSEWeb: a programmer assistant for reusing open source code on the web," in *Int'l Conf. Automated Softw. Eng.*, 2007, pp. 204–213.

[21] Y.-C. Wu, L. W. Mar, and H. C. Jiau, "CoDocent: Support API usage with code example and API documentation," in *Int'l Conf. Softw. Eng. Advances*, 2010, pp. 135–140.

[22] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei, "MAPO: mining and recommending API usage patterns," in *European Conf. Object-Oriented Prog.*, 2009, pp. 318–343.

[23] H. Zhong, L. Zhang, T. Xie, and H. Mei, "Inferring resource specifications from natural language API documentation," in *Int'l Conf. Automated Sofw. Eng.*, 2009, pp. 307–318.