Automatic Parameter Recommendation for Practical API Usage

Cheng Zhang[1]*, Juyuan Yang[2], Yi Zhang[2], Jing Fan[2], Xin Zhang[2], Jianjun Zhao[2]*, Peizhao Ou[2]
[1]*Department of Computer Science and Engineering,* [2]*School of Software*
Shanghai Jiao Tong University, China
{cheng.zhang.stap, zhao-jj}@sjtu.edu.cn, {juyuanyang, yizhang1990.mail, j.fan, hyperzh, peizhao.ou}@gmail.com

***Abstract*—Programmers extensively use application programming interfaces (APIs) to leverage existing libraries and frameworks. However, correctly and efficiently choosing and using APIs from unfamiliar libraries and frameworks is still a non-trivial task. Programmers often need to ruminate on API documentations (that are often incomplete) or inspect code examples (that are often absent) to learn API usage patterns. Recently, various techniques have been proposed to alleviate this problem by creating API summarizations, mining code examples, or showing common API call sequences. However, few techniques focus on recommending API parameters.**

In this paper, we propose an automated technique, called Precise, to address this problem. Differing from common code completion systems, Precise mines existing code bases, uses an abstract usage instance representation for each API usage example, and then builds a parameter usage database. Upon a request, Precise queries the database for abstract usage instances in similar contexts and generates parameter candidates by concretizing the instances adaptively.

The experimental results show that our technique is more general and applicable than existing code completion systems, specially, 64% of the parameter recommendations are useful and 53% of the recommendations are exactly the same as the actual parameters needed. We have also performed a user study to show our technique is useful in practice.

***Keywords*-recommendation; API; argument; parameter; code completion**

I. Introduction

Application programming interfaces (APIs) are extensively used in software development to reuse libraries and frameworks. Since there are numerous APIs providing various functionalities, developers are likely to encounter unfamiliar APIs in their work. Unfortunately, APIs are generally difficult to learn, due to various factors, such as complexity of the application domain and inadequate API design [20]. As a result, developers often have to make considerable effort to learn how to use APIs correctly and efficiently.

To alleviate this problem, a number of techniques [11], [21], [15] have been proposed to facilitate the usage of APIs. Among these, a highly automated approach is a code completion system, which promptly provides developers with programming suggestions, such as which methods to call and which expressions to use as parameters. Traditional code completion systems generally propose their suggestions based on type compatibility and visibility. Such suggestions may become insufficient for complex frameworks, where some types have too many methods or fields to suggest. Therefore, some recent work [9] tries to improve the suggestions via mining API usage data from code bases.

Most existing techniques focus on *choosing the right method to call*. However, as discussed in relevant studies [9], [18], it is a non-trivial task to *choose the right (actual) parameter(s) for a method call* in an API usage. Table I shows the statistics of API declarations and usages in the code of Eclipse 3.6.2 [4], Tomcat 7.0 [1], and JBoss 5.0 [5].

Table I
STATISTICS ON METHOD DECLARATIONS AND INVOCATIONS

Program	param declaration	non-param declaration	param invocation	non-param invocation
Eclipse 3.6.2	64%	36%	57%	43%
JBoss 5.0	58%	42%	60%	40%
Tomcat 7.0	49%	51%	60%	40%
average	**57%**	43%	**59%**	41%

[1]"param" and "non-param" are abbreviations for "parameterized" and "non-parameterized", respectively.

Table I shows that 57% of the method declarations are parameterized, that is, the methods are passed one or more parameters when being called. In accordance with the statistics of method declarations, 59% of the method invocations actually have parameters. Furthermore, about 50% of these actual parameters cannot be recommended by existing code completion systems, because the expected parameters are too complex to recommend[2]. Sometimes, even if an existing code completion system can provide correct recommendations, it can be difficult to find the right parameter from a large number of parameter candidates.

Figure 1 shows an example of choosing an API parameter using Eclipse JDT. In the example, the developer has to choose the right actual parameter (i.e., `IHelpUIConstants.TAG_DESC`) from over **50** string variables and literals. Unfortunately, the field `TAG_DESC` is even not visible in Figure 1, as it is ranked too low by dictionary order. More importantly, since the code completion system does not suggest that interface `IHelpUIConstants` should be used, the developer probably has to learn this

*Corresponding authors.

[2]We use the Eclipse JDT code completion system as a typical example. The details will be explained in Section III-A

usage from code examples or documentations. Therefore, it would be really helpful if the code completion system could provide more informative recommendations of API parameters and rank them in a more sophisticated way (e.g., suggest using `IHelpUIConstants` and put `TAG_DESC` near the top of the list).

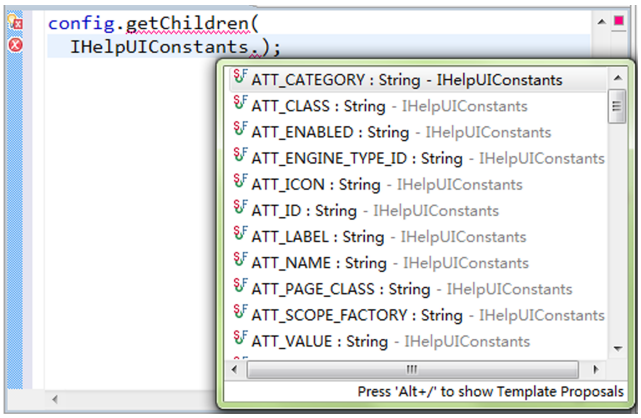

Figure 1. An example of parameter recommendation by Eclipse JDT.

Besides slowing down the development process, unfamiliarity with parameter usage may even harm the correctness of programs. Bug 153895 in the bug database of Eclipse[3] is a typical example of the bugs caused by incorrect usage of API parameters. The bug reporter suggested that invocations to method `getBestActiveBindingFormattedFor` of interface `IBindingService` should be used as the actual parameters of two method invocations. In the end, the developer took the suggestion and fixed the bug by replacing the incorrect parameters with the correct ones. It is worth noting that existing code completion systems can provide little help in this case, because such method invocations are too complex to recommend (also see Section III-A).

In this paper, we propose *Precise*, an automated approach to **p**arameter **rec**ommendation for API **u**sag**e**, which is able to recommend the kinds of API parameters that are frequently used in practice but mostly overlooked by existing code completion systems. During programming, when the developer has already selected a method and is about to determine the actual parameter(s) of the method, Precise automatically recommends a concise list of well sorted parameter candidates. The basic idea of Precise is to extract usage instances from existing programs and adaptively recommend parameters based on these instances and the current context. On receiving a request for recommendations, Precise first uses k-nearest neighbor (k-NN) [23] queries on the usage database built from a large code base to find abstract usage instances of parameters that are commonly used in similar contexts. Then Precise generates concrete recommendations based on the abstract usage instances. Precise ranks its recommendations with respect to the similarity of contexts and the frequency of usage, helping developers select the right parameters more easily. Two heuristic rules are used to reduce the search space of parameters, making Precise practical and efficient. Experimental results show that Precise can recommend parameters with good accuracy under strict constraints on the number of recommendations. Specifically, the expected parameters are included in the top 10 recommendations in 53% of the cases, and in 64% of the cases Precise provides useful hints to help choose the right parameters. The main contributions of this paper are:

1) Precise, an automatic parameter recommender, which improves existing code completion systems. To the best of our knowledge, it is the first automatic technique focusing on parameter recommendation.
2) We have implemented Precise and performed an experiment to show its effectiveness. Combined with Eclipse JDT, the implementation provides useful recommendations in 67% of the cases. We have also conducted a user study to confirm its usefulness.

The rest of this paper is organized as follows. Section II shows an empirical study on the usage of API parameters. Section III describes the technical details of Precise. Section IV shows the evaluation results. Section V compares Precise with closely related work. Section VI concludes this paper and discusses our future work.

Table II
STATISTICS ON EXPRESSION TYPES OF ACTUAL PARAMETERS

Expression Type	Eclipse	JBoss	Tomcat	average
simple name	47.35%	38.79%	39.97%	42.04%
method invocation	12.16%	11.96%	10.44%	11.52%
qualified name	11.28%	4.33%	4.28%	6.63%
field access	1.15%	0.57%	0.32%	0.68%
array access	1.58%	1.09%	0.63%	1.10%
cast expression	1.16%	1.28%	0.37%	0.94%
string literal	4.71%	21.71%	19.74%	15.39%
number literal	3.44%	5.66%	4.04%	4.38%
character literal	1.02%	1.29%	0.56%	0.96%
type literal	0.42%	0.68%	2.24%	1.11%
boolean literal	3.83%	2.05%	2.13%	2.67%
null literal	1.81%	1.67%	1.72%	1.73%
total percentage	89.91%	91.08%	86.44%	**89.14%**

II. API PARAMETER USAGE IN PRACTICE

A major challenge in parameter recommendation is that there can be too many possible parameter candidates that are type compatible with the expected parameter. For example, when we try to recommend the actual parameter for the method `m(int a)`, there are an almost infinite number of parameter candidates (of type `int`), including all the integer literals, accessible integer variables, method calls returning integers, etc. In addition, the parameter can be an arithmetic expression, such as `(a+b*c)`, which may have a very complex structure. Therefore, as a fundamental strategy of Precise, we propose two heuristic rules to focus the approach on parameters of a limited number of structures. The underlying assumption is that *most of the parameters belong to a small number of types of structural patterns*.

[3]https://bugs.eclipse.org/bugs/show_bug.cgi?id=153895

In order to support the assumption, we have done an empirical study to see how API parameters are used in practice by looking into three subjects, namely Eclipse 3.6.2, JBoss 5.0, and Tomcat 7.0. These are large-scale programs that are widely used in real-world settings. Thus the result of our study is supposed to be reasonably representative. As shown in Table II, over 89% of the actual parameters of APIs can be classified into 12 types of expressions. Figure 2 gives a brief description of the expression types and their structures. The expression types are formally described in Eclipse JDT documentation [3] which is in line with the Java language specification [7]. Several kinds of expressions are out of the scope of the 12 expression types, including array creation, assignment, infix expression, etc. They together make up 11% of the actual parameters in the subjects.

⟨method invocation⟩ ::= [⟨expr⟩"."]⟨identifier⟩"("⟨arguments⟩")"
⟨field access⟩ ::= ⟨expr⟩"."⟨identifier⟩
⟨qualified name⟩ ::= ⟨name⟩"."⟨simple name⟩
⟨array access⟩ ::= ⟨expr⟩"["⟨expr⟩"]"
⟨cast expression⟩ ::= "("⟨type⟩")"⟨expr⟩
⟨literal⟩::=⟨string literal⟩ | ⟨number literal⟩ | ⟨character literal⟩
| ⟨type literal⟩ | ⟨boolean literal⟩ | ⟨null literal⟩
⟨simple name⟩ ::= ⟨identifier⟩
⟨expr⟩ ::= ⟨method invocation⟩ | ⟨field access⟩ | ⟨qualified name⟩
| ⟨array access⟩ | ⟨cast expression⟩ | ⟨literal⟩ | ⟨simple name⟩
⟨arguments⟩ ::= [⟨expr⟩{","⟨expr⟩}]
⟨name⟩ ::= ⟨qualified name⟩ | ⟨simple name⟩

Figure 2. Grammar of expression types. Precise focuses on the first six expression types, except boolean literal and null literal. Note that this grammar just shows the structures of expression types and the accurate grammar can be found in relevant documentations [3], [7].

III. Approach

The work flow of Precise consists of two phases. In the first phase, Precise builds a usage database by analyzing parameter usage instances and their contexts in a code base. In the second phase, for a recommendation request, Precise queries the database using the context of the request as the key and retrieves a group of abstract usage patterns. Then Precise concretizes and sorts the recommendations.

The overall approach is similar to that of the work on API method recommendation [9], where static features are used to capture contexts and several algorithms, including a specialized k-NN algorithm, are used to calculate the relevance of the recommended methods. Similarly, Precise also uses static features to capture contexts and uses a k-NN algorithm for relevance calculation. Nevertheless, since Precise targets the recommendation of parameters (instead of methods), it uses different static features which focus on the code prior to the parameters (see Section III-B2). In addition, the characteristics of parameter recommendation have motivated us to design a different evaluation (see Section IV) and propose two heuristic rules based on the empirical study in Section II. These rules govern both phases of Precise, making the technique practical while still being effective. In this section, we first introduce the rules and then describe the two phases in detail.

A. Heuristic Rules for Search Space Reduction

Rule of expression type. The first rule is to restrict the two phases to the parameters of certain expression types, namely method invocation, field access, qualified name, array access, cast expression, and some types of literals. Specifically, the usage database does not include any parameter whose expression type is unspecified in the rule, and such a parameter is not generated as a recommendation either. The grammar of the focal expression types is shown in Figure 2.

While focusing Precise on these expression types, we leave out three of the 12 types in Table II, namely simple name, boolean literal and null literal. Since parameters of these expression types are recommended by Eclipse JDT by default, Precise does not take them into account. As shown in Table II, the focal expression types of Precise take up about 43% of all the actual parameters. More importantly, parameters of these expression types are generally more difficult for developers to find or compose than simple names, boolean literals and null literals. Precise also does not deal with expression types that are not shown in Table II, because they are infrequently used.

Rule of structural complexity. Within the expression types specified in the first rule, there can still be some parameters that are too complex to recommend. For example, an actual parameter can be a method invocation in the form of `array[0].getFoo().getName().substring(3)`. Also, such complex parameters are infrequently used and thus can hardly be amenable to Precise which relies on existing examples. Therefore, as the second rule, we restrict the two phases to the parameters (expressions) whose structural complexity is less than or equal to **3**.

The *structural complexity* of an expression, `exp`, is the number of leaf nodes in the abbreviated abstract syntax tree of `exp`. The *abbreviated abstract syntax tree* of `exp`, is derived from the abstract syntax tree of `exp` by removing every node (and its sub-tree) that represents the argument list of a method invocation or the array index of an array access. Additionally, literals are treated as a leaf node, instead of being further decomposed syntactically.

This rule essentially confines Precise to the parameters whose structures are reasonably simple. According to our study on Eclipse 3.6.2, JBoss 5.0 and Tomcat 7.0, the structural complexity of nearly 95% of the actual parameters is less than or equal to 3, which means that Precise can possibly recommend the right parameters in most cases.

B. Building Parameter Usage Database

Following the heuristic rules, Precise first analyzes programs in a code base to build a usage database. Conceptually the usage database stores the accumulated data about which parameters are used for an API in a specific context. In a recommender system for methods (e.g., [9]), it is straightforward to use the method name, (fully qualified) declaring type, and argument type list to represent a method.

In contrast, Precise requires a certain degree of abstraction in representing the actual parameters, in order to make the data useful in the subsequent phase.

1) Abstract Representation of Parameters: The need for abstract representation of parameters stems from the fact that, in different cases, some variables are named differently, though they essentially represent the same entity. In Figure 3, either `b` in case (a) or `button` in case (b) represents a reference to an instance of `Button`. Suppose the recommender system learns from case (a) and tries to recommend a parameter for method `setAttrib` in case (b), it will fail to find a match if its learning and recommendation are based on exact variable names. In this example, the most useful information is the structure of the expected actual parameter and the type of the base variable, that is, the actual parameter should be a method invocation to `getAttrib()` and the base variable is of type `Button`.

Precise abstracts the parameter usage instances before storing them into the usage database, in order to retain the essential information, while pruning unnecessary details (e.g., variable names). During the abstraction, Precise first identifies the structures of the actual parameters and categorizes the parameters into several expression types, such as literal, method invocation, and field access. Then, for each expression that is not a literal, Precise resolves the type of its sub-expressions that are local variables and replaces the variable names with their types. In Figure 3, the actual parameter used in case (a) will be represented by `Button.getAttrib()`, as the variable `button` is replaced by its type `Button`.

Case (a): code snippet from code base:

```
1  public class ExampleClass{
2    public void addButton(Panel panel){
3      Button b = new Button();
4      b.setVisible(true);
5      b.setAttrib(0);
6      panel.init();
7      panel.addElement(b);
8      //use p1 to denote b.getAttrib() here
9      panel.setAttrib(b.getAttrib());
10   }
11 }
```

Case (b): code snippet under development:

```
1  public void methodToDevelop(){
2     Button button = new Button();
3     this.setAttrib(?);
4  }
```

Figure 3. Example Java code.

2) Defining Parameter Usage Contexts: Contextual information enables recommender systems to make recommendations adaptively. Precise uses four static features to capture the context in which each actual parameter is used. The features are extracted from the generally available context (i.e., the code prior to the parameter), because the code after the parameter has usually not yet been written when Precise is triggered. In software maintenance scenarios where developers mainly make edits to existing code, we conservatively assume that the code after the parameter will probably be changed and thus unsuitable for representing the context. We use the case (a) in Figure 3 to illustrate the features. In the code snippet, we suppose that the actual parameter of interest is the method invocation at line 9 (i.e., `b.getAttrib()`), and we use $p1$ to denote it hereafter.

Feature 1: the signature of the formal parameter bound to the actual parameter. This feature represents the most specific usage context for an actual parameter. The signature of a formal parameter is the combination of the method signature, the parameter type, and the position in the parameter list. In Figure 3, the value of feature 1 of $p1$ is `Panel::setAttrib::1(int)`.

Feature 2: the signature of the enclosing method in which the actual parameter is used. This feature represents the contextual information on the surrounding scope (of the method). As discussed in [9], such a feature is useful to identify the usage context in the case of overriding or implementing a method defined in a super type. Currently this feature just includes the method signature, ignoring the method's declaring type. It is a strategy to save the effort to explore type hierarchies at the cost of the accuracy of context. In Figure 3, the value of feature 2 of $p1$ is `addButton(Panel)`.

Feature 3: the method invocations that have happened on the variable used in the actual parameter. For the kinds of parameters Precise focuses on, a variable `v` can be the base variable of a method invocation (e.g., if the actual parameter is `v.getA()`) or the qualifier of a field access (e.g., if the actual parameter is `v.fieldA`). This feature captures the history of the variable `v` before it is used in the actual parameter. Such a history is useful to identify specific patterns of "preparing" a variable for the parameter usage. We use the method signature to denote each method invocation. Thus, in Figure 3, the value of feature 3 of $p1$ is the list of methods invoked on `b`, that is, `<<init>(), setVisible(boolean), setAttrib(int)>`. We use `<init>()` to represent the constructor of a class.

Feature 4: the method invocations that have happened on the base variable of the method invocation using the actual parameter. Similar to feature 3, this feature is designed to capture the history of a variable, but the variable here is not used in the actual parameter, instead it is an essential part of the method invocation using the parameter. As shown in [9], such a feature can effectively represent the context of method usage. Therefore, we use this feature to include method-related context into the parameter usage context. Again we use the method signature to denote each method invocation. In Figure 3, the value of feature 4 of $p1$ is the list of methods invoked on `panel`, that is, `<init(), addElement(Object), setAttrib(int)>`.

3) Transformation and Representation of Parameter Usage Instances: When building the usage database, Precise first scans the code base to find every parameter usage instance. Then it extracts the feature values for each parameter usage instance, while abstracting the actual parameter. Each instance of parameter usage consists of both the feature values and the abstract parameter representation. In order to support the k-NN algorithm (described in Section III-C), Precise transforms each parameter usage instance into a vector before storing it into the parameter usage database. The transformation algorithm is shown in Algorithm 1.

Algorithm 1: Usage Instance Transformation

Input: a List $iList$ of InstanceData for each FormalParameter
Output: a Table $result$

```
1  result.name = feature1;
2  foreach InstanceData d of iList do
3      d.features = d.features − feature1;
4      foreach feature f ∈ d.features do
5          put f.values into a Set named featureValues;
6      end
7  end
8  foreach InstanceData d of iList do
9      int[ ] value = new int[featureValues.size() + 1];
10     initialize value with 0;
11     d.features = d.features − feature1;
12     foreach feature f ∈ d.features do
13         foreach value v ∈ f.values do
14             find the index of v in Set featureValues;
15             value[index] = 1;
16         end
17     end
18     int usageIndex = indexUsage(d.getParam());
19     value[features.size()] = useageIndex;
20     add value into result;
21 end
22 return result;
```

In the first place, we believe that it is generally useless to recommend an actual parameter for a method by learning from parameter usage instances of other methods or other formal parameters of the same method. Therefore, at the beginning of the transformation, we create one table in the usage database for each unique formal parameter, using feature 1 to generate the table name. All the actual parameters bound to a specific formal parameter will be processed and stored in the table corresponding to that formal parameter. Then within a table, we compute the *range* for features 2, 3, and 4, which is the set containing all the feature values of the usage instances in the table (lines 2–7, Algorithm 1). Each element in the range corresponds to a column in the table. When a usage instance is being transformed, the value of a column is set to 1 if the column's corresponding value occurs in the instance's feature values; otherwise the column is set to 0 (lines 9–17, Algorithm 1). The last column in the table is designed for storing the (abstract) actual parameter. Since the information of an actual parameter is generally too rich to be stored as plain text, we keep a map between each actual parameter and its unique index and store the indices in the last table column (lines 18 and 19, Algorithm 1, where the method `indexUsage` indexes each actual parameter and maintains the map between actual parameters and their indices). The example in Figure 4 shows the feature values of $p1$ and three other contrived instances and their encoded vectors stored in the usage database.

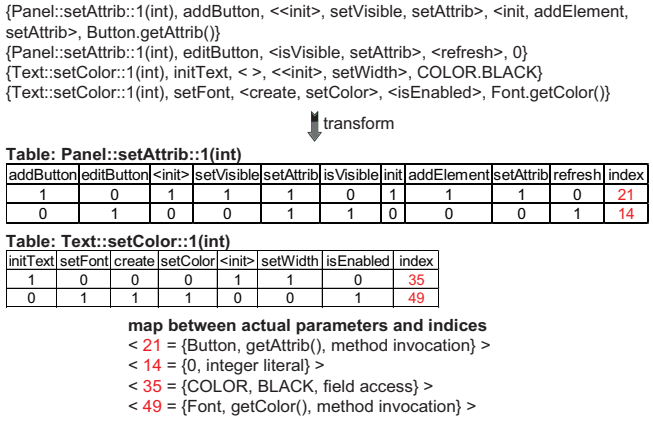

Figure 4. Example usage instances and their representations in the usage database. For brevity, the parameter types of method signatures are omitted.

C. Generating Parameter Candidates

1) Finding Similar Usage Instances: The k-NN algorithm is commonly used to classify an instance based on the "nearest" training instances (i.e., the instances with known classes or categories). Precise uses the information of the nearest neighbors to guide its recommendation. When being requested for a recommendation, Precise queries the usage database using k-NN to find k usage instances whose contexts are the most similar to the context from which the request is issued. Then the type and structure information of these similar usage instances is used to generate a list of parameter recommendations. The details of recommendation generation will be described in Section III-C2.

Similar to a parameter usage instance, a request context is also represented by feature values. However, such a context does not include the actual parameter. For a request for recommendation, its value of feature 1 is used to determine which table will be queried for the nearest parameter usage instances. To represent the request context , features 2 and 4 can be computed in the same way as those of the parameter usage instances in the usage database. However, since the actual parameter is yet to be recommended, it is unclear which variable is used in the parameter. Thus we have to take into account all the accessible variables for computing feature 3. Here accessible variables include local variables, formal parameters of the enclosing method, and fields. Thus features 2 and 4 are common for all the accessible variables. For each accessible variable, `v`, we will compute its history of method invocation (feature 3) and combine it with features 2 and 4 to create an individual context, called *variable context* (and `v` is called the *context base* of the variable context). We will also generate an individual context, called *non-variable context*, which only includes

features 2 and 4 so as to recommend literal parameters. Therefore, for a given request, if n variables are accessible, then there are $n+1$ contexts for representing the request's possible contexts. For each of the $n+1$ contexts, we find the k nearest instances in the usage database. In order to perform the k-NN search, we first transform the request context into a vector with respect to the table for the formal parameter. The way of evaluating each vector element is the same as in the transformation of a parameter usage instance, except that the vector of a request context does not contain a usage index. Then we define the distance between a request context and a parameter usage instance based on their vector forms:

Definition The distance, $distance(rc, pi)$, between a request context, rc=$\langle c_1^{'}, c_2^{'}, ..., c_n^{'} \rangle$ and a parameter usage instance, $pi = \langle c_1, c_2, ..., c_n, index \rangle$, is calculated by $distance(rc, pi) = \sqrt{\sum_{i=1}^{n}(c_i - c_i^{'})^2}$, where $c_i^{'}$ and c_i are the encoded feature values of the request context and the parameter usage instance, respectively, and $index$ is the usage index.

Based on the above definition, we find the k nearest neighbors for each of the $n+1$ contexts. In the k-NN search, we first set k to 1 and increment k until the number of concrete recommendations reaches a threshold or all the instances in the usage database are used. Currently the threshold is 10. Then, for each context, we categorize the parameter usage instances with the same parameter index into one group. For each group, we get a summary instance that consists of the abstract parameter (corresponding to the usage index) and the number of instances in that group (called *frequency*). If there are m different usage indices in the nearest neighbors (m may vary between different contexts), we will get m summary instances.

2) Concretizing Parameter Recommendations: When the summary instances have been extracted from the usage database, Precise has gained some knowledge about the parameter usages in similar contexts. As described in Section III-B1, such knowledge is abstract, in that it is mainly about the type and structure information of the actual parameters. Precise takes a further step to generate concrete recommendations based on the summary instances.

In recommendation concretization, Precise takes different strategies with respect to different types of expressions for different contexts. If the abstract parameter of a summary instance of a non-variable context is a literal, then Precise directly uses the literal as the parameter recommendation. Otherwise, if the abstract parameter of a summary instance of a variable context contains a type name that is abstracted from a variable, Precise checks whether the context base is type-compatible with the type in the abstract parameter. If so, Precise generates a parameter recommendation by replacing the type name with the context base (i.e., the accessible variable). In this case, other segments in the parameter recommendation, such as method names or package names, are copied verbatim from their counterparts in the abstract usage instance. For example, in case (b) of Figure 3, if the abstract parameter `Button.getAttrib()` is a part of a summary instance of the variable context with context base `button`, Precise will generate a recommendation `button.getAttrib()`.

After recommendation generation, we sort the recommendations in descending order by the frequencies of their corresponding summary instances. Then we sort the recommendations with the same frequency in dictionary order with respect to their parameter expressions. Last, the sorted recommendations are presented to the user. Currently we insert the recommendations by Precise into the original list of parameter recommendations by Eclipse JDT and place them on the top of the list.

IV. Evaluation

We have implemented Precise as an Eclipse plug-in[4]. At the front end, the implementation seamlessly extends Eclipse JDT to present parameter recommendations. At the back end, the implementation leverages the DOM/AST API provided by the Eclipse JDT platform to perform source code analysis, and uses Weka [13] to do the k-NN search. Tables in the usage database are stored as ARFF files (i.e., Weka's native input format), and the map between indices and actual parameters is stored in XML files. These two parts, which are correlated by the usage indices, play the role of the usage database in Precise.

A. Objective Experiment

Performance measure. Using the implementation, we have performed an objective experiment to evaluate the usefulness of Precise. For each request, while Precise normally generates a list of parameter candidates, there is exactly one expected actual parameter which we call the *target*. The target may or may not be included in the recommended parameters. In the experiment, we impose a constraint on the number of parameter candidates, that is, we only check the top **10** candidates if there are more than 10 of them. By controlling the total number of recommended parameters, we can evaluate Precise's usefulness by checking how often it provides useful recommendations, without being concerned about how many recommendations it generates each time. We choose 10 as the threshold, because the user may have a context switch [19] if she has to scroll down the list to check the recommendations lower than 10 with default JDT configurations. We assume that all the top 10 recommendations attract equal attention from the user, because they are presented in a short list that can be quickly checked.

We determine whether Precise is successful for a specific request by checking whether the list of parameter recommendations includes the expected actual parameter. Thus we use $Recommendations_{made}$ to denote *the total number of times that Precise tries to recommend a list of parameters*

[4]The source code and screen shot of the implementation and experimental data are available on http://stap.sjtu.edu.cn/~chengzhang/precise/

and $Recommendations_{successful}$ to denote *the number of times that the list includes the expected parameter*. Then we use the precision (defined as below) to represent Precise's performance of parameter recommendation. (Since there is exactly one target for each request, $Recommendations_{made}$ is equal to *the number of targets* in the experiment. Thus the precision can also be viewed as the recall.)

$$precision_{successful} = \frac{Recommendations_{successful}}{Recommendations_{made}}$$

However, as in the case shown in Figure 1, even if the recommendations fail to include the exact actual parameter, they can still provide useful information, such as which class should be used as a part of the parameter. More specifically, a recommended parameter is said to be *partially correct*, if it correctly indicates 1) the base variable of a method invocation, 2) the qualifier of a qualified name, or 3) the method name of a method invocation. The partial usefulness is proposed based on the following observations:

- In cases 1 and 2, as shown in Figure 1, when a remote entity is needed, it is helpful to suggest the base variable or qualifier, which can reduce the search space greatly.
- Case 3 occurs mostly due to polymorphism, where Precise reveals the usage pattern of the method, leaving limited uncertainty of choosing the right base variable.

In order to take into account such cases, we use $Recommendations_{useful}$ to denote *the number of times that the recommendation list provides either completely or partially correct information with respect to the expected actual parameter*. Then we define another precision to measure Precise's performance from this viewpoint.

$$precision_{useful} = \frac{Recommendations_{useful}}{Recommendations_{made}}$$

Subjects and validation strategy. Since Precise is proposed to facilitate the use of framework APIs, it is reasonable to evaluate Precise on individual frameworks. In the experiment, we focus on the SWT framework [6] and use all the projects using SWT in the code base of Eclipse (Classic 3.7) as the subjects. We choose the SWT framework because it provides a large code base containing about $1,270,536$ lines of code. Moreover, SWT is a specialized framework for GUI development. Compared with common libraries, such as the JDK API, SWT is less familiar to developers and has more specific patterns of parameter usage. Therefore, developers are more likely to benefit from an enhanced code completion system for the SWT framework.

We take the strategy of 10-fold cross validation. More specifically, the whole data set is evenly divided into 10 subsets at random. In each iteration of validation, one subset is used as test data , while the other nine subsets are used as training data for building the usage database. Ten iterations are performed so that each subset is used as test data once. We split the parameter usage instances based on class, that is, all the parameter usage instances occurring in a Java class are used together in either the test set or the training set. During the experiment, each actual parameter in the test set of an API method defined in SWT is used to test Precise. When being used as a test instance, the actual parameter is "hidden" and a request is issued to Precise. All the test instances must be of the expression types that Precise focuses on. The recommendations for the request are checked against the actual parameter to see whether they are successful or useful.

Table III
RESULTS OF THE OBJECTIVE EXPERIMENT

No.	#Param	#Req	#Success	#Useful	p_s	p_u
1	3095	978	546	89	56%	65%
2	3785	1154	599	157	52%	66%
3	3252	984	516	123	52%	65%
4	2967	931	495	102	53%	64%
5	2932	950	443	107	47%	58%
6	3504	1094	575	105	53%	62%
7	3030	928	562	110	61%	72%
8	3191	994	513	88	52%	60%
9	3275	1043	530	106	51%	61%
10	3313	996	501	140	50%	64%
Avg.	3234	1005	528	113	**53%**	**64%**

[5]As we use 10-fold cross validation, there are 10 rows of data. In the columns, '#Param' means the number of actual parameters in the test set; '#Req' means the number of recommendation requests; '#Success'and '#Useful' represent the number of times when the recommendations contain the expected actual parameter and useful information, respectively; p_s and p_u stand for $precision_{successful}$ and $precision_{useful}$, respectively.

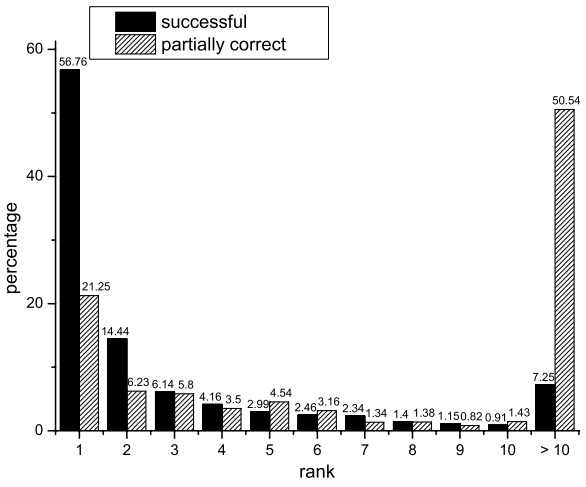

Figure 5. Rank distributions of recommendations.

Results. Table III shows the experimental results. The average $precision_{successful}$ and $precision_{useful}$ are 53% and 64%, respectively. It means that Precise can often provide useful recommendations and sometimes the right parameter is included in the list of recommendations. It is worth noting that the kinds of parameters recommended by Precise are not recommended by the original Eclipse JDT and they are generally more complex and difficult to find or compose. Therefore, the results can be viewed as promising.

Since the experiment focuses on the top 10 recommendations, it is interesting to study the effectiveness of Precise's ranking strategy. As shown in Figure 5, about 57% of the successful recommendations are placed at the top 1 of the list (and 93% in the top 10). By comparison, only 21% of the partially correct recommendations are ranked top 1 (and 49% in the top 10). We have analyzed the result data and found that a large number of low-ranked partially correct recommendations are generated for some common methods (the limitation of Precise on common methods will be discussed later). An indication of Figure 5 is that there might be large room for improvement in $precision_{useful}$ by designing better ranking strategies.

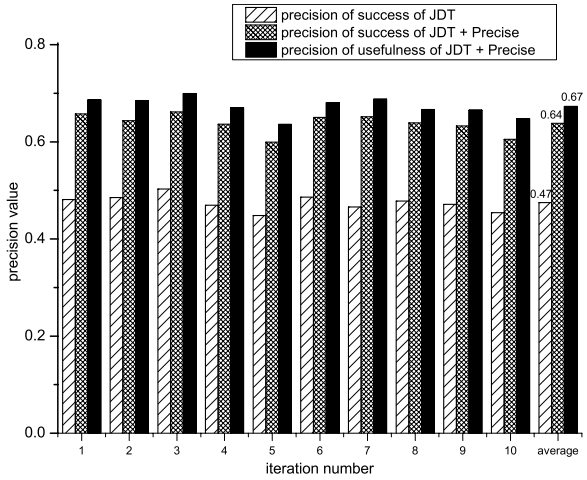

Figure 6. Precision values of Eclipse JDT with and without Precise.

.

Because the recommendations actually presented to the user also include the recommendations by Eclipse JDT, from the perspective of users, we are more interested in how much improvement Precise has made to the Eclipse JDT code completion system. The Eclipse JDT code completion system generates its parameter recommendations mainly based on type-based rules. The recommendations include limited (simple) types of expressions. By default, the Eclipse JDT code completion system is activated every time a parameterized method is selected by the user. Whenever the expression type of the expected parameter is in the scope of the JDT code completion system, it will certainly be included in the list of recommendations (although there may be more than 10 parameters recommended). However, due to its limitation on expression types, the JDT code completion system will certainly fail if the expected parameter is complex. As shown in Figure 6, the $precision_{successful}$ is 47%. As the combination of JDT and Precise, the enhanced code completion system has an average $precision_{successful}$ of 64%. Moreover, its average $precision_{useful}$ is 67%, indicating that it can often provide useful recommendations.

We have compared Precise with four of its variants, named *F1+Freq*, *F1+F2+Freq*, *F1+F3+Freq*, and *F1+F4+Freq*. In F1+Freq, usage instances are selected from corresponding database tables (using feature 1) and then sorted by frequency. In contrast to Precise, F1+Freq does not use k-NN search. We choose such a variant for comparison, because it represents a typical frequency-based solution and is more reasonable than some naive frequency-based approaches (e.g., recommendation based on the frequency of type-compatible actual parameters used for any API method). In F1+F2+Freq, we augment F1+Freq with k-NN search only based on feature 2. F1+F3+Freq and F1+F4+Freq are similarly designed with different features. In essence, Precise can be named *F1+F2+F3+F4+Freq* using this convention.

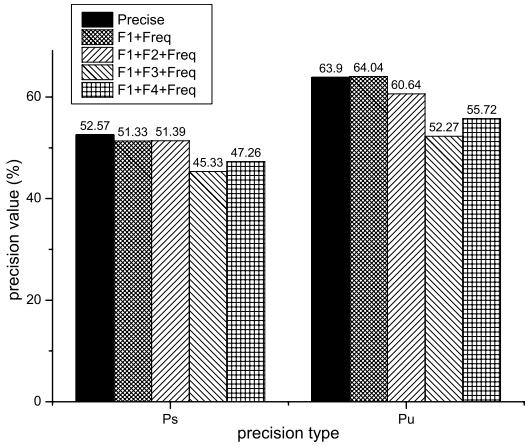

Figure 7. Precision values of several variants of Precise.

As shown in Figure 7, Precise has the highest p_s and almost the highest p_u. Although the difference between Precise and F1+Freq seems insignificant, we find that features 2, 3, and 4 can indeed capture the context information to make Precise outperform F1+Freq in several cases. It indicates that it is necessary to include extra features (besides feature 1) in order to enable adaptive recommendation, while the current features probably still need improvement. To our surprise, the results show that adding a single feature, especially feature 3 or 4, does not improve (or even harms) the performance. By analyzing the data, we find that an individual feature is often unable to accurately capture useful context information, but it may make the instance vectors high dimensional (e.g., for feature 3, there can be a number of calls on the variable used in the actual parameter). As a result, the distances computed using such vectors may fail to capture the contextual similarity correctly. Since Precise has the best performance, the result indicates the necessity of combining the features and the need for improving each of them. We also find that the difference in performance between Precise and the four variants is not so significant, because there are often fewer than 10 recommendations for a request. In this case, Precise and the other four approaches succeed (or fail) together. The root cause of such cases is the scarcity of abstract usage instances in the usage database.

Although we use large training sets, we need more programs using SWT to get more conclusive results.

A concern on the usability of Precise is its runtime, because its second phase has user interactions. To study the responsiveness of Precise, we have measured the runtime of the second phase on a desktop with a 2.40GHz dual-core CPU and 4GB memory and also that of the first phase on a server with a 2.33GHz quad-core CPU and 16GB memory. As expected, most of the runtime (793 seconds on average) is taken by the first phase. By comparison, the second phase takes little runtime (76 milliseconds on average), which is presumably negligible. Therefore, we believe that Precise can be seamlessly integrated into the Eclipse JDT code completion system, using pre-built usage databases. In the user study, we have further investigated this issue.

Limitations. An important assumption of Precise is that it can learn usage patterns from the code base. However, some APIs (e.g., `get(Object)` in `java.util.Map`) are widely used in numerous contexts. These APIs are so common that their actual parameters are too diverse for Precise to learn any usage pattern. Although SWT is a specialized framework, there are still several common methods in it. A typical example is the method `setText(String)` defined in classes `Button`, `Label`, etc. The method is used to set the text on a widget. It is imaginable that various strings will be used as the actual parameter of `setText`. As a result, Precise has extremely bad performance for this method, often recommending a large number of string literals and variables without success. In the experiment, `setText` takes up 32% of the requests for recommendations and the $precision_{successful}$ and $precision_{useful}$ of Precise for `setText` are 21% and 24%, respectively.

As a learning-based approach, Precise has little chance of recommending the right parameters, if there are few usage instances with similar contexts in the training data. This limitation is actually a major reason for Precise's failures observed in some iterations of the 10-fold cross validation. In the experiment, the expected actual parameter is sometimes a method or a field that is used very locally (e.g., a private field is only used in its declaring class). Meanwhile, we split the parameter usage instances based on class. When the usages of such a method or field are concentrated in a small number of classes in the test set, they are out of reach of Precise. To overcome the limitation in recommending for unseen usage patterns, we will try to improve Precise with generative techniques (e.g., [12]).

B. User Study

We have invited eight participants to use the Precise plug-in to finish two small programming tasks. The participants are students with more than three years of experience of Java and at least two years of experience of Eclipse on average. Thus they are familiar with the Eclipse JDT code completion system. The two programming tasks are adapted from the examples of the SWT framework. One task is to implement a simple panel with various fields and texts, while the other is to implement a simple browser. We deleted several statements from the original examples, leaving skeletons of the programs. In this way, the participants can concentrate on finishing the programs using APIs from SWT, without making design-level efforts, such as creating new interfaces or classes. After completing the tasks, each participant was asked to fill in a questionnaire which consists of five questions. For each question, a participant first chose a score from 5 (strong agreement), 4 (weak agreement), 3 (neutral), 2 (weak disagreement), and 1 (strong disagreement), and then gave a detailed explanation of the choice. The questions and the summary of the answers are described as below.

Q1: Did Precise propose the right parameters? (average: 3.875, median: 4) This question is designed to confirm that the users can recognize the successful recommendations. The resulting scores and the explanations show that the participants are able to identify the right parameters in most cases. This is partly due to the short list of recommendations provided by Precise (i.e., up to 10 extra items besides those recommended by JDT). The scores also indicate that the participants have perceived the effectiveness of Precise in accordance with the $precision_{successful}$ (64%), that is, it is useful, although not perfect.

Q2: When you could not find the right parameter in the recommendations, did you feel Precise still gave useful hints? (average: 3.875, median: 4) This question investigates whether the partially correct information can indeed be useful. We almost get the same positive answers as in Q1. A participant said:"*Some of the parameters are 'dot-outcome', and I really feel it's convenient*". While confirming the partial usefulness, the answer indicates that Precise probably focuses on the right expression types.

Q3: Did Precise correctly rank the parameters? (average: 3.5, median: 3) This question checks the ranking strategy of Precise. The scores show that the participants do not think Precise has ranked the recommendations well. We have investigated the explanations and found that some participants just did not pay attention to the ranking, because they focused on finding the right parameter in the list. The purpose of the ranking is to put the most relevant recommendations at the top of the list. However, users usually wonder why such a sophisticated ranking (instead of dictionary order) is used, especially when they fail to find the right parameters and try to use the partial information.

Q4: Did Precise speed up your development compared to the default Eclipse code completion? (average: 3.375, median: 4) This question attempts to investigate whether the improvement by Precise can be noticeable in the whole Eclipse system. The answers are generally positive, indicating that Precise is promising to be useful in practice. However, the answers are far from conclusive, because various factors may affect the final results. By checking

one participant's program, we found that she mostly failed to find the right methods to call. As a result, Precise was not activated as expected. We believe that the integration of novel code completion techniques (e.g., API recommendation [9] and Precise) could improve the existing system more significantly than individual techniques.

Q5: Is Precise well integrated into Eclipse? (average: 4.28, median: 4) This question investigates whether the runtime overhead of Precise is acceptable in practice. Among the seven participants who gave answers, three of them chose the score of 5. Four participants expressed the feeling that Precise worked as a natural part of Eclipse JDT. The answers have confirmed that Precise does not cause perceivable slowdown when proposing recommendations.

In summary, the user study has confirmed the usefulness of Precise shown by the objective experiment. However, it has also revealed an inadequacy of Precise: its ranking strategy lacks explanations and better integration with other techniques is needed.

C. Threats to Validity

A threat to the evaluation result is that the usefulness of Precise may not be accurately measured by the precisions. We have performed the user study to alleviate this problem. However, since the sample is small, the result still needs further validation. One threat to the generalization of the result is that the evaluation was performed on a specialized framework. As discussed in Section IV-A, common APIs can have adverse effects on Precise's performance. Further studies on the applicability of Precise to diverse frameworks may be necessary. Due to the limitation of resources, the programming tasks used in the user study are relatively small and simple. It is unclear whether the same result can be obtained on real-world programming tasks which are larger and more difficult. We are working on a stable version of the Precise plug-in and hope to integrate it into the Eclipse Code Recommenders [2]. With a number of real users, we may be able to obtain more conclusive results.

V. Related Work

API recommendation. Bruch et al. [9] propose Intelligent Code Completion Systems (ICCS) which learns from existing programs to recommend API methods. ICCS searches the code bases for API calls under the similar context and generates proposals using a customized k-NN algorithm. Similarly, Precise uses data mining techniques to make recommendations. However, while ICCS exclusively recommends method calls, Precise aims to predict actual parameters for method calls. Therefore, Precise can be seen as an extension to ICCS. Robbes and Lanza [19] use program history to improve code completion systems. They model the program history as a sequence of changes and propose a series of code completion algorithms mainly based on the method-level and class-level changes. Their work provides code completion for classes and methods and thus is different from ours. Hou and Pletcher [14] enhance the code completion system by introducing new features such as grouping, sorting and filtering. These features enable the completion system to reduce the number of API candidates and rank them in more sophisticated ways, making it easier for programmers to find the right APIs. Their approach essentially tries to highlight the most relevant APIs when there are too many of them to choose. In contrast, Precise can still be helpful when programmers have to type in actual parameters (instead of choosing them). Moreover, Precise recommends API parameters, not APIs. Other techniques [16], [11], [15], [17] have also been proposed to facilitate the use of APIs. Different from Precise, they do not focus on parameter recommendation and code completion.

Code search. Code search engines enable developers to learn the usage of unfamiliar APIs by providing code examples. Bajracharya et al. [8] create SAS which applies a grouping and indexing algorithm to search the repository and shows relevant code snippets. Besides the text search used by SAS, structural information of program source code is also used in code search engines, such as SNIFF [10]. SNIFF is more relevant to Precise in that it makes use of type information to refine its result set. Other approaches (e.g., [22]) also leverage search engines to facilitate the reuse of frameworks. Although search-based approaches effectively help API learning, they still require developers to pick from the list of search results the right methods or parameters to be placed in their particular contexts. In contrast, in Precise, this task is performed automatically. In Precise, recommendations are proposed in place by the code completion system, possibly saving some effort of extracting pieces of information from examples and mapping them to the right places. Moreover, Precise focuses on structural API parameters, instead of general code examples.

VI. Conclusion and Future Work

We have proposed Precise to automatically recommend actual parameters for APIs. By extracting usage instances from existing programs, Precise provides useful parameter recommendations with satisfactory precisions, showing its ability to improve the state-of-the-art code completion systems. In our future work, we are planning to study how to use program evolution information to improve Precise. Moreover, we will try generative techniques to recommend parameter usages that are unseen in the code base.

Acknowledgment

We are grateful to the anonymous reviewers for their suggestions to improve the paper and to Sai Zhang for his insightful discussions. This work was supported in part by National Natural Science Foundation of China (NSFC) grants 60970009, 60673120, and 91118004.

REFERENCES

[1] Apache Tomcat. http://tomcat.apache.org/.

[2] Eclipse Code Recommenders. http://www.eclipse.org/recommenders/.

[3] Eclipse JDT API Specification, Eclipse documentation. http://help.eclipse.org/helios/index.jsp.

[4] Eclipse Project. http://www.eclipse.org/.

[5] JBoss Application Server. http://www.jboss.org/jbossas/.

[6] SWT: The Standard Widget Toolkit. http://www.eclipse.org/swt/.

[7] The Java Language Specification, Third Edition. http://docs.oracle.com/javase/specs/jls/se5.0/html/j3TOC.html.

[8] Sushil Bajracharya, Joel Ossher, and Cristina Lopes. Sourcerer: An internet-scale software repository. In *Proceedings of the 2009 ICSE Workshop on Search-Driven Development-Users, Infrastructure, Tools and Evaluation*, SUITE '09, pages 1–4, Washington, DC, USA, 2009. IEEE Computer Society.

[9] Marcel Bruch, Martin Monperrus, and Mira Mezini. Learning from examples to improve code completion systems. In *Proceedings of the the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT symposium on The Foundations of Software Engineering*, ESEC/FSE '09, pages 213–222, New York, NY, USA, 2009. ACM.

[10] Shaunak Chatterjee, Sudeep Juvekar, and Koushik Sen. Sniff: A search engine for Java using free-form queries. In *Proceedings of the 12th International Conference on Fundamental Approaches to Software Engineering: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009*, FASE '09, pages 385–400, Berlin, Heidelberg, 2009. Springer-Verlag.

[11] Barthélémy Dagenais and Martin P. Robillard. Recommending adaptive changes for framework evolution. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 481–490, New York, NY, USA, 2008. ACM.

[12] Sumit Gulwani. Dimensions in program synthesis. In *Proceedings of the 12th international ACM SIGPLAN symposium on Principles and Practice of Declarative Programming*, PPDP '10, pages 13–24, New York, NY, USA, 2010. ACM.

[13] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. The WEKA data mining software: an update. *SIGKDD Explor. Newsl.*, 11:10–18, November 2009.

[14] Daqing Hou and David M. Pletcher. Towards a better code completion system by API grouping, filtering, and popularity-based ranking. In *Proceedings of the 2nd International Workshop on Recommendation Systems for Software Engineering*, RSSE '10, pages 26–30, New York, NY, USA, 2010. ACM.

[15] David Kawrykow and Martin P. Robillard. Improving API usage through automatic detection of redundant code. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE '09, pages 111–122, Washington, DC, USA, 2009. IEEE Computer Society.

[16] David Mandelin, Lin Xu, Rastislav Bodík, and Doug Kimelman. Jungloid mining: helping to navigate the API jungle. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '05, pages 48–61, New York, NY, USA, 2005. ACM.

[17] Hoan Anh Nguyen, Tung Thanh Nguyen, Gary Wilson, Jr., Anh Tuan Nguyen, Miryung Kim, and Tien N. Nguyen. A graph-based approach to API usage adaptation. In *Proceedings of the ACM international conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 302–321, New York, NY, USA, 2010. ACM.

[18] Michael Pradel and Thomas R. Gross. Detecting anomalies in the order of equally-typed method arguments. In *Proceedings of the 20th International Symposium on Software Testing and Analysis*, ISSTA '11, pages 232–242, 2011.

[19] Romain Robbes and Michele Lanza. Improving code completion with program history. *Automated Software Eng.*, 17:181–212, June 2010.

[20] Martin P. Robillard. What makes APIs hard to learn? answers from developers. *IEEE Softw.*, 26:27–34, November 2009.

[21] Jeffrey Stylos, Brad A. Myers, and Zizhuang Yang. Jadeite: improving API documentation using usage information. In *Proceedings of the 27th international conference Extended Abstracts on human factors in computing systems*, CHI EA '09, pages 4429–4434, New York, NY, USA, 2009. ACM.

[22] Suresh Thummalapenta and Tao Xie. Parseweb: a programmer assistant for reusing open source code on the web. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated Software Engineering*, ASE '07, pages 204–213, New York, NY, USA, 2007. ACM.

[23] Ian H. Witten and Eibe Frank. *Data Mining: Practical Machine Learning Tools and Techniques, Second Edition*. Morgan Kaufmann, 2005.