Exploring API Method Parameter Recommendations

Muhammad Asaduzzaman, Samiul Monir, Chanchal K. Roy and Kevin A. Schneider
Department of Computer Science
University of Saskatchewan, Saskatoon, Canada
Email:{md.asad, samiul.monir, chanchal.roy, kevin.schneider}@usask.ca

***Abstract*—A number of techniques have been developed that support method call completion. However, there has been little research on the problem of method parameter completion. In this paper, we first present a study that helps us to understand source code localness to method parameters. Based on our observations, we developed a recommendation technique, called *Parc*, that leverages source code localness property to collect parameter usage context. *Parc* uses previous code examples together with contextual and static type analysis to recommend method parameters. Evaluation of our technique against the only available state-of-the-art tool using a number of subject systems and different Java libraries shows that our approach has promising potential. We also explore the parameter recommendation support in the Eclipse Java Development Tool (JDT).**

***Index Terms*—Automatic Code Completion, Method Parameter Recommendations, Recommendations, API**

I. Introduction

Developers use framework and library APIs to reuse code during software development. This not only speeds up development but also saves time and resources. However, studies have shown that it is difficult to learn APIs due to various factors, such as inadequate examples and documentation [27], [29]. It is also difficult to remember APIs due to their sheer volume. To alleviate these problems, modern integrated development environments (IDEs) contain a code completion feature. It has been found that code completion is one of the top ten commands used by developers [26]. It speeds up the process of writing code by reducing typos or other programming errors, and frees the developer from remembering every detail. For example, as a developer types a method call, a code completion system typically uses autocomplete popups to recommend a set of method names, and the developer can then select the appropriate method call from the list. Although a number of techniques have been developed, most focus on the problem of suggesting method calls and leave the task of completing method parameters to the developers. However, determining the correct parameters to call a method is a non-trivial task and requires more attention [3]. Unless otherwise specified, in this paper we use the term parameter to refer to the actual parameter.

Incorrect use of API method parameters can lead to software bugs [19], [21] or can cause runtime exceptions. For example, the *valueOf* method in the Java String class returns the string representation of the parameter. The *valueOf* method has been overloaded to accept different forms of parameters and the overloaded method is chosen according to the static type of the parameter. One of the overloaded methods takes an Object as a parameter and the other takes a char array (char[]). The first method contains a check for the null value but the other one does not. The call String.valueOf(null) will execute the second overloaded method and throw a null pointer exception since type char[] is more specific than Object according to the Java language specification. However, casting the parameter to Object selects the first overloaded method. Since this method has a check for null value, the method will not throw an exception [24]. Although developers can identify and solve the problem by searching online and reading documentation, the effort is considerable. A parameter recommendation technique can help avoid this extra effort by suggesting the required casting operation.

There has been very little research on the problem of automatic parameter completion; however, Zhang et al. [3] have developed a parameter completion tool: *Precise*. Precise mines previous code examples to collect parameter usage patterns. Given a request for parameter completion, the tool uses a K-nearest neighbour algorithm to recommend proposals by matching the similarity of the current context with the past parameter usage examples. Despite the contribution of Precise, we see a gap between their work and what we can do regarding parameter completion. In this paper, we follow on from their work to further explore the problem and to improve support for parameter completion.

Unlike the study by Zhang et al. [3], we start with a manual investigation to understand how developers complete method parameters. During our study we found that parameter usages are *locally specific*. For example, before using an array access expression as a method parameter, developers typically instantiate the array close to that parameter position. When a method invocation is used as a method parameter, other method calls and language constructs that are related to that method call typically are located close together. In this paper, we conduct an exploratory study to understand parameter usages and leverage the findings to support parameter completion. In particular, we answer the following research questions:

RQ1: Is source code locally specific to parameters ?

By studying the local context of parameter usage, we hope to gain insights into building a more robust parameter completion technique.

RQ2: How can we capture the localness property to recommend method parameters?

We propose an example based code completion technique that collects parameter usage context from past examples. The technique only considers tokens close to the parameter

position. The usage context in our case consists any tokens (except identifiers, literals, braces and access specifiers) within the top four lines prior to the method call. When a developer requests parameter code completion, the technique tries to match the current usage context with that of the collected examples. It then performs static type analysis to adapt the best matched example parameter in the current development context.

RQ3: Does the technique compare well with Precise and JDT?

To the best of our knowledge, Precise is the only state-of-the-art technique for recommending API method parameters. We thus compare our technique with Precise using two large subject systems. The results from the study suggest that the technique has strong potential and can support more parameter expression types than Precise. We also investigate parameter recommendation support in Eclipse JDT. A large number of method parameters are in the simple name expression category, but Precise cannot recommend them. To compare with JDT we focus our attention to parameters of the simple name category. We ignore others because JDT has very limited or no support for them. We show that the usage patterns of simple name parameters are also locally specific and their localness can be captured in a different way. Evaluation using parameters from two different API libraries and with two different subject systems shows the effectiveness of the new technique. The modified technique has been integrated with our parameter recommendation system.

Our contributions include:

1) A study that empirically validates that source code is locally specific to method parameters.
2) A technique that leverages source code localness property, static types and previous code examples to recommend method parameters.
3) An evaluation of our proposed technique with existing state-of-the-art tools using different subject systems and with different API libraries.

The remainder of the paper is organized as follows. Section II describes related work. Section III describes important concepts related to the study. We introduce our proposed technique in Section V and summarize the evaluation results in Section VI. Section VI-B describes our technique for recommending method parameters for simple name expression types and compares the results with Eclipse JDT. We discuss some important issues about our work in Section VII. Section VIII discusses threats to the validity of our work. Finally, Section IX concludes the paper.

II. Related Work

The most relevant work to our study is that of Zhang et al. [3]. They propose a technique, called Precise, that mines existing code bases to generate a parameter usage database. A parameter usage instance consists of four pieces of information: (i) the signature of the formal parameter bound to the actual parameter, (ii) the signature of the enclosing method in which the parameter is used, (iii) the list of methods that are called on the variable used in the actual parameter, (iv) methods that are invoked on the base variable of the method invocation using the actual parameter. Given a method invocation and a parameter position, Precise identifies the parameter usage context in the current position and then looks for a match in the parameter usage database using a K-NN algorithm. Precise has been evaluated using SWT library method parameters and Eclipse. Our study differs from theirs in a number of ways. First, Precise cannot recommend parameters of the following expression types: simple name, boolean, null literal, and class instance creation. Our proposed technique can not only recommend parameters of all the above four expression types, but also those that are supported by Precise. Second, we use a different approach to construct the parameter usage context compared to Precise that takes advantage of source code localness to method parameters. Simple tokenization is suffice to collect the usage context. Finally, we also investigate the parameter recommendation support of JDT, which was not explored in the previous study.

There are also a number of other code completion techniques available in the literature. They either use previous code examples or the static type system to recommend completion proposals. However, they all focus on method call completion instead of parameter completion. Bruch et al. [9] propose the Best Matching Neighbours (BMN) completion system that uses the k-nearest neighbour algorithm to recommend method calls for a particular receiver object. Hou and Pletcher [1], [10] develop a code completion technique that uses a combination of sorting, filtering and grouping of APIs. Asaduzzaman et al. [20] develop an example-based context sensitive code completion technique that leverages locality sensitive hashing and textual similarity measures to recommend method calls. Robbes and Lanza [8] propose a set of algorithms that use program history to recommend class and method names.

There are also a number of other techniques or tools that use previous code examples, but their goals are different than ours. Nguyen et al. [6], [7] use a graph-based algorithm to develop a code completion technique. While their technique focuses on automatic completion of API usage patterns, we focus on completing method parameters. Thung et al. [23] develop a technique that given a textual description of a feature request recommends API method names for implementing a feature. The technique leverages records of previous changes made to software systems. Hill and Rideout [2] develop a technique to support automatic completion of a method body by searching similar code fragments or code clones in a code-base. Mooty et al. [5] develop an Eclipse plugin, called Calcite, that helps developers to instantiate an object of a class.

III. Preliminaries

This section describes some important concepts related to our study.

A. Parameter Completion Query

The term query indicates an incomplete method call without any parameters (see Figure 1 for an example). The goal is to

TABLE I: Examples of parameter expression types (highlighted in bold) and their distribution in three different subject systems

Parameter Expression Types	Example	Distribution of Parameters in Different Expression Types		
		JEdit	ArgoUML	JHotDraw
Array Access	button.add(**actions [i]**);	0.19	0.51	1.26
Array Creation	pd.setListData(**new Object[]**{**new LoadingPlaceholder**()});	0.14	-	0.06
Boolean Literal	frame.setVisible(**true**);	5.53	7.36	4.26
Cast Expression	progress.setMaximum((**int**)**max**));	0.48	0.68	5.57
Character Literal	list.addChar(**'A'**);	0.43	-	-
Class Instance Creation	frame.setSize(**new Dimension(100,100)**);	11.66	11.44	7
Field Access	g2d.setRenderingHints(**painter.renderingHints**);	0.07	-	0.33
Instanceof Expression	field.setEditable(**e.getItem() instanceof GlobVFSFileFilter**);	0.02	0.06	-
Method Invocation	menubar.add(**Box.createGlue()**);	16.91	16.32	12.06
Simple Name	Dimension dim= new Dimension(100, 100); frame.setSize (**dim**) ;	40.69	39.08	36.77
Qualified Name	Container c = frame.getContentPane() c.add(new JLabel("Place",**BorderLayout.CENTRE**);	8.79	9.96	17.42
Null Literal	JOptionPane.showMessageDialog(**null**, message,JOptionPane.ERROR_MESSAGE);	0.69	1.22	1.69
Number Literal	buffer.setSize(**100**);	8.09	5.85	7.46
Parenthesized Expression	rectangle.setHeight((**oldHeight*2**));	0.19	0.45	0.61
String Literal	label.setText(**"Location"**);	3.52	1.92	4.67
This Expression	clipboard.getContents(**this**);	1.84	4.30	0.68
Type Literal	SwingUtilities.getAncestorOfClass(**EditPane.class**,ta);	0.71	0.80	0.48

```
10. public void addComponent(TextArea ta){
11.     ta.setLineWrap(true) ;
12.     ta.setWrapStyleWord(true);
13.     JScrollPane scroll=new JScrollPane(ta);
14.     this.add(_
15. }
                  Incomplete Method Call
```

Fig. 1: An example of a parameter completion query

complete the parameters in order to call the method. For each query, we know the receiver type, the method name and the set of tokens that appears prior calling the method, but we do not know the actual parameter(s) to complete the method call.

B. Parameter Expression Types

An expression is a syntactic construction that can give us a value. Developers use a number of expressions types to complete method parameters. Table I shows the list of parameter expression types with examples. More about these expressions can be found in the JDT documentation [25].

C. Distribution Of Parameter Expressions

To determine the frequency of different parameter expression types we consider three different subject systems: JEdit [14], ArgoUML [15] and JHotDraw [16]. For each system we collect parameters from the API method calls of Swing and AWT libraries. Table I shows the distribution of parameters into different expression categories. Despite the difference in subject systems and API libraries, our finding is consistent with that of Zhang et al. [3]. The largest number of parameters fall in the simple name category (more than 36% in all three systems). The majority of the remaining parameters fall under the following three expression types: method invocation, qualified name and class instance creation. For JEdit, 8.79% of the parameters are of the qualified expression type and the number reaches 17.42% for JHotDraw. The method invocation expression type comes second for JEdit and ArgoUML (around 16% for both systems) and 12.06% for JHotDraw. The percentage of parameters for the class instance creation expression type ranges from 7% to more than 11%. Then come various literal expression types. In general, array access, cast expression, simple name, qualified name, method invocation, class instance creation, this expression and literal expression types (number, boolean, null and string literals) cover more than 98% of method parameters. Therefore, in this study we focus our attention on these eleven parameter expression types. We ignore others because they are difficult to autocomplete due to the complexity of the parameter expression types.

IV. RQ1: Is source code locally specific to method parameters?

We say source code is locally specific to method parameters if tokens that appear in close proximity and prior to using method parameters favor them. Then there should exist a skewed probability distribution of those tokens when we group them based on the receiver type, the method name and the parameter position. To empirically determine this we follow the procedure described by Tu et al [22] (we collect SWT API method parameters of Eclipse system). The idea is to use the entropy measure from information theory to determine the probability distribution of those tokens using the following equation: $entropy = \sum_{i=1}^{k} -p_i \log(p_i)$

Here, p_i is the probability of a token i that appears before a specific method parameter over all examples of that parameter position that is followed by token i. The more the entropy measure is close to zero, the higher the distribution will be skewed. On the contrary, the closer the value is to $\log_2 k$ (k is the number of examples of that parameter position), the more the probability distribution of those tokens will be uniform.

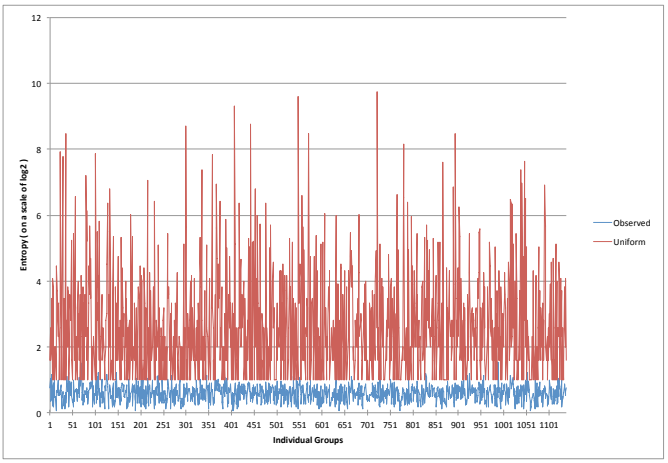

Fig. 2: Mean entropy distribution for the top ten tokens when we group code examples based on receiver type, method name and parameter position.

For each method parameter we determine the top n tokens (in this experiment we set the value of n to 10) before calling the method and then group them based on the receiver type of the method call, the method name and the parameter position. For each group, we determine the entropy of each token using the above equation and then report the mean entropy value. Figure 2 shows the results along with the entropy measure for uniform distribution. We can see from the figure that the observed mean entropy values are way smaller than that of uniform distributions. This confirms that we can use locally specific tokens to recommend method parameters. It should be noted that we did not consider the method name in the top ten tokens because of its uniform probability distribution in each group of examples.

V. RQ2: How can we capture the localness property to recommend method parameters?

To answer the above question we describe our proposed technique to recommend method parameters. Before discussing details of the technique we summarize challenges in recommending method parameters. First, static type systems cannot help much simply because there are a large number of variables whose type matches with the expected type of the parameter. There can also be a number of methods whose return type matches with the expected type of the parameter. Furthermore, the type of an actual parameter can be a subtype of the formal parameter. Second, a parameter of a method can take expressions of different categories (such as simple name, method invocation, class instance creation etc.). Third, some parameters are difficult to predict because of the high degree of variability or complex structures in its parameter usage examples which makes the recommendation challenging (such as postfix, prefix and infix expression types). However, they are only a few in number and we ignore them in this study.

To support automatic completion of API method parameters, we develop a technique that collects parameter usage instances from previous code examples to form a database, and we call this *Parc*. When a developer requests for a parameter completion, our technique determines the requested parameter usage context and the candidate list. The list is then sorted using the similarity between the context of the requested parameter with that of parameters stored in the database. Finally, the top few candidates are recommended as completion proposals after removing duplicates. In the following section we describe each step in detail.

A. Building the parameter usage database

To build the parameter usage database, we walk through code examples to look for API method invocations. Then for each method parameter we collect the following pieces of information:

1) **Method Name**: The name of the method containing the parameter.
2) **Receiver Type**: The fully qualified name of the receiver type of that method.
3) **Parameter Position**: The position of the parameter in the method parameter list.
4) **Feature**: Features capture the localness property of method parameters. A feature consists of a set of tokens that appears within the top k lines prior calling a method with parameters and also located within the same method that encloses that method with parameter. We only consider any method names, any class or interface names and any keywords because these tokens show the most skewed probability distribution. After considering various values of k we obtain the best performance of our technique using $k = 4$.
5) **Generic Representation**: The generic string representation of a parameter. The identifiers appeared in parameter examples can have different names, but they refer to the same parameter. The generic representation is used to identify duplicity in the parameter recommendations. To generate the generic representation, we replace any simple name that appeared in the parameter expression with its type qualified name. For example, consider the following method invocation where the parameter is a simple name: panel.add(button). The generic representation would be javax.swing.jbutton. Consider another case where the parameter is a method invocation: tab.setText(button.getText()). The receiver type of the parameter method invocation here is javax.swing.jbutton and the generic representation would be: javax.swing.jbutton.getText

The above items represent a parameter usage instance as shown in Figure 3. We use an indexing scheme where the parameter usage instances are indexed based on the method name, parameter position and receiver type. The usage context of a parameter is stored by concatenating all of the terms where any two consecutive terms are separated by a single space.

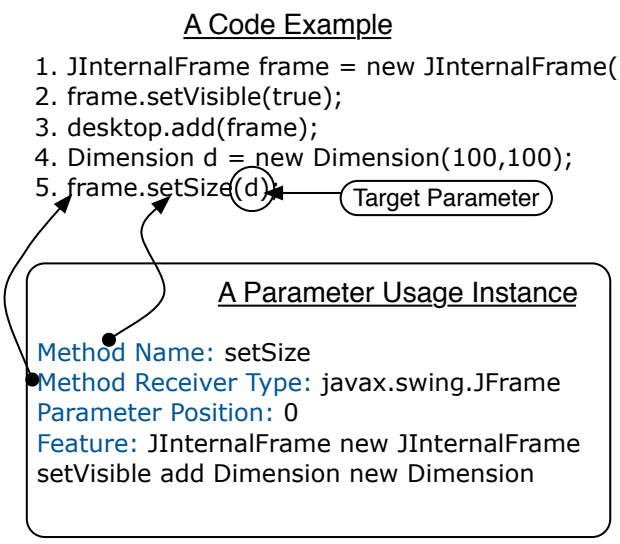

Fig. 3: An example of a parameter usage instance

B. Collect features from query

When a developer requests for a method parameter recommendation (in Eclipse this can be pressing ctrl + space after writing the method name), *Parc* collects the same set of information it collected while building the parameter usage database, except generic representation of the parameter since we do not yet know actual parameter. We refer to this information as a *query instance* and the corresponding feature as a *query feature*. We use the method name, receiver type and parameter position value of the query instance as an index to locate all previous parameter usage instances. We call this set the mapping candidates.

C. Determine feature similarity

For each mapping candidate and the query pair, we apply cosine similarity to determine their feature similarity and then sort the mapping candidates based on the descending order of the similarity value. If two mapping candidates have the same feature similarity with the query, we use the frequency of the mapping candidate to break the tie. Determining cosine similarity with all mapping candidates can be very time consuming. Therefore, we introduce an intermediate step. We use a form of locality sensitive hashing to calculate usage context similarity using similarity preserving hash values instead of calculating textual similarity. Since the hashing technique converts a large string into a much smaller bit sequence, the similarity can be calculated very quickly.

We use the *simhash* algorithm [18] to generate 128 bit binary hash values, also known as *simhash* values. The technique has been found effective in detecting duplicate web pages [17]. The more similar two string are, the more smaller would be the Hamming distance between their *simhash* values. The Hamming distance between two binary values is equal to the number of ones in their bitwise exclusive OR operation. The smaller the Hamming distance is, the closer the two *simhash* values are. Therefore, after determining the Hamming distance we sort the parameter candidates in ascending order of distance value and select the top 500 parameter candidates, which we refer to as *likely parameter candidates*. We now determine the cosine similarity of the query context with each likely parameter candidate. We then sort the likely parameter candidates in descending order of similarity value and use their generic representation to remove any duplicates.

D. Static analysis and recommendation

At this point, we have a sorted list of parameters. Before making any recommendation we need to validate whether the parameter matches with the current context. For this reason, we use the following set of rules to create parameter recommendations:

1) If the parameter is a literal type, we directly use that for the recommendation.
2) If the parameter is a method invocation expression or a qualified name, we need to consider two different cases. If the receiver is empty or a type variable, we use the parameter without any change. However, if the receiver is a simple name, we first find all variables that are type compatible with the receiver variable and within the scope of the query method call (see Section VI-B for detail about how we generate this list of variables). We then replace the receiver with the topmost variable and insert the method invocation expression in the recommendation list.
3) If the parameter is a simple name, we search the query context to look for variables that are within the same scope as the query method call and type compatible with the formal parameter. This time we insert the top three variables in the list of recommendations. We limit the insertion of up to three variables because there can be a large number of type compatible variables that reside within the scope of query context and we found the best result using that value.

After generating the recommendations, they are placed on top of the JDT completion proposals to present to the users. The number of recommendations generated by the *Parc* is configurable by users.

VI. RQ3: Does the technique compare well with Precise and JDT?

To answer the above question we evaluate *Parc* with existing parameter recommendation techniques (this includes both Precise and Eclipse JDT) we use two different subject systems: Eclipse 3.7.2 [12] and NetBeans [13]. Both systems are large in size and have a long development history. Since our technique focuses on API method parameter completion, we use two different API libraries. For the Eclipse system, we collect all parameters of SWT library methods. For the NetBeans system, we collect method parameters of Java Swing and AWT libraries. Our selection of libraries is based on the fact that all these libraries are frequently used by developers for developing

applications. Thus, automatic parameter completion support for those API methods is more likely to help developers.

We apply the ten-fold cross validation technique to measure the performance of each technique [11]. First, we divide the entire data set into ten different folds. Next for each fold, we use code examples from the nine other folds to train the technique for parameter completion. The remaining fold is used to test the technique. To make the result comparable with Precise, the folds are generated based on classes. This means that all the parameter usage instances occurring in a class are either used together for training or for testing.

We provide parameter usage instances occurring in the training set to the parameter completion techniques for training. During testing, for each API method parameter we ask a code completion technique to generate completion proposals. The actual parameter used in the test set is hidden from the techniques, but the code appears prior to that parameter is available to them. After generating the completion proposals, we check whether the target parameter appears within the top ten recommendations.

We use precision and recall to measure the performance which are defined as follows:

$$Precision = \frac{recommendations\ made \cap relevant}{recommendations\ made} \quad (1)$$

$$Recall = \frac{recommendations\ made \cap relevant}{recommendations\ requested} \quad (2)$$

Here, *recommendations made* is the total number of time a parameter completion technique recommends parameters. The term *relevant* refers to the total number of time the actual parameter is present in the top few recommendations. The term *recommendations requested* denotes the number of parameters in our test data.

A. Evaluation Results

This section presents results of our evaluation. We were interested to see how *Parc* performs in predicting all eleven parameter expression types. Table II shows precision and recall values for two different subject systems and for eleven different parameter expression types. For Eclipse, Parc has a 47.65% precision for the top position and a 72.06% precision for the top ten positions. The recall value is also very high. For the top ten positions *Parc* achieves more than a 70% recall value. For the NetBeans system *Parc* shows consistent performance. For the top ten positions precision is 72.06% and recall is nearly 70%. However, Precise did not perform well in this study. This is because Precise cannot detect parameters of the following expression types: simple name, null literal, boolean, and class instance creation. However, a large number of method parameters are of simple name and class instance creation expression types.

We only include Precise in this table to show that it cannot detect a large number of method parameters but they can be easily detected by *Parc*. Since JDT can recommend method parameters of simple name and the highest number of parameters fall in that category, it could easily achieve higher precision and recall value than Precise in this experiment. This can hide the important fact that such parameters are easier to detect and JDT cannot recommend any complex parameters that are detected by Precise. While one can combine JDT with Precise by appending completion proposals of JDT after the recommendations from Precise, there are challenges in using them together. It may be the case that the actual parameter is a simple name but Precise recommends other parameter expressions. Appending completion proposals after Precise indicates that we will definitely miss the parameter in the top positions. Thus, we exclude JDT from this experiment and focus on comparing with Precise only.

TABLE II: Evaluation results of parameter recommendation techniques for all eleven parameter expression types *Parc* can detect

Subject System	Recom.	Precision		Recall	
		Precise	Parc	Precise	Parc
Eclipse	Top-1	11.75	47.65	11.07	46.65
	Top-3	15.26	65.05	14.38	63.68
	Top-10	18.45	72.26	17.38	70.73
NetBeans	Top-1	16.67	46.46	13.78	44.86
	Top-3	22.10	66.20	18.27	66.75
	Top-10	25.46	72.06	21,04	69.57

To determine how *Parc* performs considering only those parameter expression types that are supported by Precise, we conduct the experiment again. This time we consider the following parameter expression types: qualified names, method names, string literals, number literals, this expressions, array access and cast expressions. Table III shows the evaluation results. For both systems, *Parc* achieves better results than Precise. For the Eclipse system and for the top position, *Parc* obtains 2% better precision value and 2.35% better recall value. While for the top ten positions *Parc* achieves 53.49% precision value, Precise obtains 51.46%. *Parc* also obtains slightly higher recall value than Precise. Although the performance of *Parc* is slightly better than Precise, these are the most difficult parameters to predict. Moreover, we exclude parameters of class instance creation type from this experiment that can be detected by *Parc* but not by either JDT or Precise. For the NetBeans system, the relative improvements become more significant. For example, for the top position *Parc* achieves a 25.87% higher precision value and a 33.16% higher recall value compared to Precise. Even for the top ten positions *Parc* performs significantly better than Precise. However, *Parc* provides more accurate recommendations and shows consistent performance across different subject systems. Table IV shows the accuracy of correctly predicted parameters across different parameter expression types for the top ten recommendations for this experiment.

B. Exploring Parameter Recommendations of Eclipse JDT

Despite the fact that a large number of method parameters fall in the simple name category, Precise cannot recommend them. However, the default code completion system of Eclipse JDT can recommend those parameters. However, Eclipse JDT

TABLE III: Evaluation results of parameter recommendation techniques using only those parameter expression types that are supported by Precise

Subject Systems	Recommen.	Precision (%)		Recall	
		Precise	Parc	Precise	Parc
Eclipse	Top-1	32.77	34.77	30.69	33.04
	Top-3	42.58	46.29	30.88	43.98
	Top-10	51.46	53.49	48.19	48.48
NetBeans	Top-1	25.82	51.69	26.06	49.30
	Top-3	34.23	70.99	34.55	67.71
	Top-10	39.42	78.38	39.79	74.75

TABLE IV: Accuracy of correctly predicted parameters for different expression types for the top ten recommendations

Subject Systems	Parameter Expression	Test Cases	Accuracy(%)	
			Precise	Parc
Eclipse	Qualified Name	470	42.12	43.82
	Method Invocation	405	47.16	46.67
	String Literal	33	39.39	36.36
	Number Literal	79	65.82	65.82
	This Expression	36	55.56	55.56
	Array Access	24	66.67	66.67
	Cast Expression	1	100	100
NetBeans	Qualified Name	860	26.86	93.02
	Method Invocation	490	29.59	54.28
	String Literal	85	47.05	50.64
	Number Literal	441	74.60	66.67
	This Expression	69	79.71	79.71
	Array Access	5	40.00	20.00
	Cast Expression	11	9.09	9.09

provides very limited or no support for recommending parameters for other expression types. In this section, we first summarize the parameter recommendation strategy of Eclipse JDT and then conduct an experiment to evaluate the technique with our proposed one for the simple name parameters only.

JDT collects local variables, parameters of the enclosing method, class variables (also known as fields) and inherited variables whose type match with the expected type of the target method parameter. We refer to this set of variables as the candidate set. Depending on the expected types of the method parameters, JDT also adds different literals to the candidate set. For example, if the expected type of the method parameter is boolean, boolean literals *true* and *false* are added. If the expected type is an object of a class, JDT adds *null* literal to the candidate set. Integer literal *0* is added when the expected parameter is a number. It then sorts the elements of the candidate set using the following sequence of rules:

- local variables have higher priority than class variables and class variables have higher priority than inherited class variables.
- A longer case insensitive substring matches of the variable name with that of the method formal parameter will prevail.
- variables that have not been used have a higher priority than those that have already been used. This rule tries to avoid recommending the same variable to multiple method parameters.
- The more closely a variable is declared to the method parameter position, the more its priority will be. Closeness is calculated using its location in the source code.

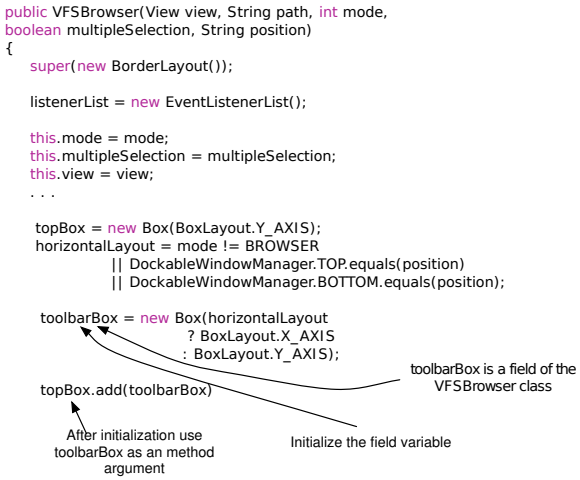

Fig. 4: When recommending a method parameter, Eclipse JDT puts the local variables first. The field variables are positioned after the local variables and method parameters. Thus, in this case JDT will place the variable toolBarBox in the last position of the completion popup. However, instead of the declaration point, considering the initialization or most recent assignment point prior to calling the target method can help us to place the variable in the top position.

TABLE V: The percentage of correctly predicted method parameters for the simple name expression category

Subject Systems	Category	JDT (%)	Parc (%)	Relative Improvent
Eclipse	Local	70	94.67	24.67
	Parameter	90.22	87.45	-2.77
	Field	62.78	66.55	3.97
NetBeans	Local	79.22	87.26	8.04
	Parameter	91.54	88.29	-3.25
	Field	24.33	29.93	5.60

We randomly select a number of examples where the method parameter is a simple name and manually investigate them. We notice that developers tend to declare a variable in the same code block the method call is located and then use that as a method parameter. The more closer a type compatible variable is declared, the higher the possibility of using the variable as a method argument. That is why the first and the last rules are more effective than the other two rules. However, there are also a number of exceptions to this parameter usage pattern. We observe cases where developers declare a list of Component variables, initialize them and then add them to a container in the same order they are declared. Situation can be worse when there are a large number of variables that are type compatible with the parameter type. In that case, JDT fails to guess the correct method parameter within top three positions.

We observe two interesting patterns. First, a developer

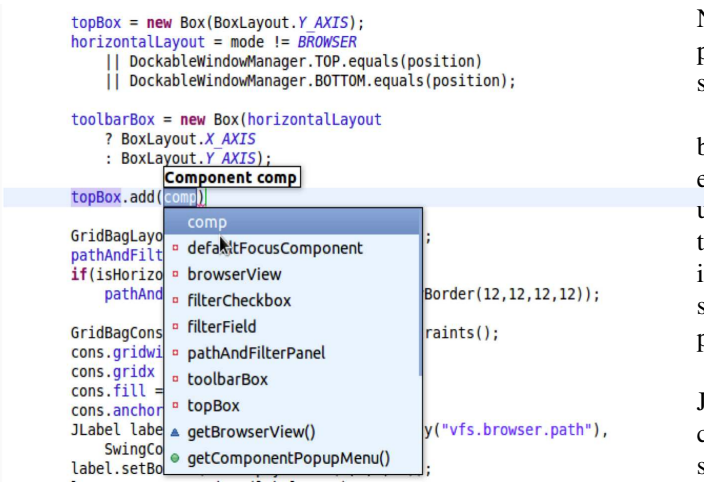

Fig. 5: An example of parameter recommendation by Eclipse JDT for the topBox.add() method. The actual parameter toolbarBox is placed at the sixth position by JDT. In case there are more type compatible local variables, JDT would place toolbarBox parameter in a more lower position. Our proposed technique can guess the parameter in the top position.

declares a list of local variables at the beginning of a method body. However, they initialize the variable just before using them as a method argument. Second, often developers initialize or assign a new value to the field variable and in the next statement they use the variable as a method parameter (see Figure 4). Since, Eclipse JDT puts field variables after the local variables, it fails to guess many field variables as a method parameter within the top positions. In both cases, instead of the declaration location, the recent initialization location of a variable can help us better predict the correct method parameter (see Figure 5).

We were interested to see whether the above findings can help us to improve parameter completion results. We change the sorting rules as follows (we refer to this as the modified approach): The more closer a variable is declared, initialized or assigned new values to the method parameter the higher the priority of the variable would be. Then there will be the unused parameters of the enclosing method, any unused class variables and finally, any unused inherited field variables. We use the term *unused* to refer to those class variables, inherited field variables and enclosing method parameters that are not initialized or assigned any new value prior to calling the target method in its enclosing method.

To evaluate our proposed sorting mechanism with that of the Eclipse JDT, we develop two different programs. Both programs parse each source file to identify the location of each method argument of simple name category and perform static analysis to generate the candidate set. However, they sort the candidate variables in two different ways. The first program imitates the sorting mechanism of JDT and the second program uses our proposed sorting rules. We identify the location of the target variable in the sorted list of candidates in both cases and record the results. For testing, we again use Eclipse 3.7.2 and NetBeans as subject systems. For the first system we consider parameters of SWT library method calls and for the second system we consider Swing/AWT library method parameters.

If the usage context of a requested method parameter best matches with that of a simple name parameter in code examples, *Parc* collects and sorts the type compatible variables using the modified approach described above. Next, it suggests the top most variable as a method parameter. Thus, it is important to improve the result for the top position. Table V shows the percentage of correctly predicted simple name parameters for the top position.

In general, our modified approach performs better than JDT for the top position. For example, for the local variable category, the relative improvement is 24.67% for the Eclipse system. We also observe improvement for field variables. Although our proposed change did not result in good results for the parameters, the relative improvement for the local variables is much higher than the relative performance decline for the parameter category. Moreover, the number of field variables and the number of cases developers use a field variable as a method parameter are much higher than the corresponding values for the parameter category. We also observe similar result for the NetBeans.

VII. Discussion

A. Why did Precise not perform well?

We further investigated to determine why Precise did not perform well with the NetBeans system. One of the reasons that contribute to the poor results of Precise is that in many cases it only partially detects the target parameter. For example, if the expression type of the parameter is a method invocation, Precise detects the name of the method invocation correctly but fails to identify the receiver of that method invocation correctly or vice versa. Consider that Precise finds a match of the query context with that of a method parameter in the training code example, where the expression type of the parameter is a method invocation (i.e., frame.getContentPane()) with simple name as the receiver (the type of the receiver is javax.swing.JFrame). Precise collects all simple names of the same type within the scope of the query (consider there are three simple names: f, myFrame, frame), substitute the receiver with each simple name and recommends all of them (f.getContentPane(), myFrame.getContentPane(), frame.getContentPane()). However, it fails to determine which simple name in the current context is deemed for the receiver of the parameter expression. *Parc* on the contrary replace the receiver with only that simple name that it finds most probable.

B. Runtime Performance

The time required for a recommendation is an important concern of the usability of any code completion system. We have measured the runtime of *Parc* in recommending completion proposals on a desktop computer equipped with a Core i7 CPU and 10 GB of memory. On average *Parc* requires 45 milliseconds to recommend completion proposals, which is considerably negligible. This indicates that *Parc* can be easily

TABLE VI: Cross-project prediction results of *Parc* under two different settings

Recommendatons	Precision (%)		Recall(%)	
	A	B	A	B
Top-1	14.52	35.49	14.01	34.57
Top-3	31.00	52.51	29.90	51.16
Top-5	38.74	59.47	37.38	57.94
Top-10	44.30	69.06	42.75	67.28

integrated with the Eclipse JDT code completion system and also leaves the opportunity to include addition type analysis. We are currently working on the implementation of *Parc* as an Eclipse plugin.

C. Cross-Project Prediction

If developers want to use our code completion system at an early stage of their project, it would be difficult to train the system due to lack of code examples. This problem can be solved by using code examples from other projects. We were interested to find whether we can apply *Parc* in a project by training code examples from other projects. This is referred to as cross-project prediction. Cross-project prediction can be difficult due to the differences in project structure, development teams, and programming rules. To perform cross-project prediction we use two different settings: (A) we train *Parc* using code examples from NetBeans, ArgoUML and JFreeChart for the SWT and Swing libraries. The test cases are collected from the JEdit system. (B) We repeat the same test but this time we allow code examples from JEdit (except those we use for testing) also to train *Parc*. As time passes and a project becomes mature, more code examples will be available. The second setting imitates this scenario. It should noted that we consider in this experiment all eleven parameter expression types *Parc* can detect.

Table VI shows the result of our cross-project prediction. When *Parc* does not have any knowledge about the test project (Setting A), it achieves 44.30% precision and 42.75% recall value for the top ten positions. Reducing the number of recommendations also penalizes the performance considerably. For example, the precision and recall values are around 14% for the top position. When we check the result we found that this is due to the differences in the projects. Many of the parameter values are only specific to the JEdit project although there are many type compatible values exist. For example, in many cases developers use the following method invocation expression as a method parameter: *jEdit.getBooleanProperty(...)*, only specific to the JEdit system and our technique fails to recommend correct prediction in all those cases.

When we include code examples from JEdit for training (Setting B), we observe 100% improvement in the precision and recall values for the top position. For the top ten recommendations, the precision value is 69.06% and the recall value is 67.28%. This also indicates that if the training data contains parameter usage examples similar to the test cases, then *Parc* can detect them. When applying a parameter recommendation system at an early stage of a project, we possibly need to use projects similar to the test project and also add examples from the project as it grows in size.

VIII. Threats to validity

There are a number of threats to this study. First, one can argue that the results may not generalize for other systems and for different libraries. We want to point to the fact that these are popular libraries and used by various software applications. The subject systems are large in size, have long development history and also used in various other studies [9], [1].

Second, there is no public API available to collect Eclipse JDT parameter completion results. We implement the algorithm used by JDT to guess parameter proposals. Although we cannot guarantee that there is no error in our implementation, we were very careful during implementation of the algorithm. To avoid any error we have manually tested the results of our implementation with proposals made by JDT. We did not find any difference in the results during our manual inspection.

Third, there are a few parameter expression categories we ignore in this study (such as infix expression, postfix expression etc.). The reason is the variability in those expression categories that make them difficult to predict. Moreover, they represent only a small fraction of total parameters. It should be noted that our technique supports large number of parameter expression categories compared to Precise.

IX. Conclusion

Towards the goal of developing an automatic parameter recommendation system, in this paper we first conduct a study to learn how developers complete method parameters in practice. This helps us better understand parameter usage patterns. Based on our observation, we develop a technique, called *Parc*, that leverages source code localness property to capture the parameter usage context. The technique models the context by considering the four lines prior to the method invocation containing the parameter. Evaluation with a number of subject systems shows that *Parc* can recommend method parameters with consistently good performance. We also compare our proposed technique with Precise, the only available state-of-the-art parameter recommendation technique, and find satisfactory results. Moreover, *Parc* supports a large number of parameter expression types for recommendation compared to Precise. In addition, we also explore parameter recommendation support of Eclipse JDT and show a way to improve the recommendation of method parameters for the simple name category. Our study reveals that we can model the parameter usage context with limited information rather than considering various different features that may not be always available.

We are currently working on the development of an Eclipse plugin that implements the technique. We are planning to conduct a user study to identify the usefulness of our proposed technique. The code, data used in the experiment, and additional information can be found online [28].

Acknowledgements: We would like to thank Cheng Zhang for providing us the Precise tool and for giving us useful suggestions.

REFERENCES

[1] D. Hou and D. M. Pletcher, "An evaluation of the strategies of sorting, filtering, and grouping API methods for Code Completion", in Proc. ICSM, 2011, pp. 233-242.

[2] R. Hill and J. Rideout, "Automatic method completion", in Proc. ASE, 2004, pp. 228-235.

[3] C. Zhang, J. Yang, Y. Zhang, J. Fan, X. Zhang, J. Zhao, and P. Ou, "Automatic parameter recommendation for practical API usage", in Proc. ICSE, 2012, pp. 826-836.

[4] D. M. Pletcher and D. Hou, "BCC: Enhancing code completion for better API usability", in Proc. ICSM, 2009, pp. 393-394.

[5] M. Mooty, A. Faulring, J. Stylos, and B. A. Myers, "Calcite: Completing Code Completion for Constructors Using Crowds", in Proc. VLHCC, 2010, pp. 15-22.

[6] A. T. Nguyen, H. A. Nguyen, T. T. Nguyen, and T. N. Nguyen, "GraPacc: A graph-based pattern-oriented, context-sensitive code completion tool", in Proc. ICSE, 2012, pp. 1407-1410.

[7] A. T. Nguyen, T. T. Nguyen, H. A. Nguyen, A. Tamrawi, H. V. Nguyen, J. Al-Kofahi, and T. N. Nguyen, "Graph-based pattern-oriented, context-sensitive source code completion", in Proc. ICSE, 2012, pp. 69-79.

[8] R. Robbes and M. Lanza, "How Program History Can Improve Code Completion", in Proc. ASE, 2008, pp. 317-326.

[9] M. Bruch, M. Monperrus, and M. Mezini, "Learning from examples to improve code completion systems", in Proc. FSE, 2009, pp. 213-222.

[10] D. Hou and D. M. Pletcher, "Towards a better code completion system by API grouping, filtering, and popularity-based ranking", in Proc. RSSE, 2010, pp. 26-30.

[11] M. Bruch, T. Schäfer, and M. Mezini, "On evaluating recommender systems for API usages", in Proc. RSSE, 2008, pp. 16-20.

[12] "The Eclipse" http://www.eclipse.org/

[13] "The NetBeans" https://netbeans.org/

[14] "The jEdit" http://sourceforge.net/projects/jedit/

[15] "The ArgoUML" http://argouml.tigris.org/

[16] "The JHotDraw" http://sourceforge.net/projects/jhotdraw/

[17] G. S. Manku, A. Jain and A. D. Sarma, "Detecting NearDuplicates for Web Crawling", in Proc. WWW, 2007, pp. 141-150.

[18] M. S. Charikar, "Similarity estimation techniques from rounding algorithms", in Proc. STOC, 2002, pp. 380-388.

[19] M. Pradel , S. Heiniger , T. R. Gross, "Static detection of brittle parameter typing", in Proc. ISSTA, 2012, pp. 265-275

[20] M. Asaduzzaman, C. K. Roy, K. A. Schneider, Daqing Hou, "CSCC: Simple, Efficient, Context Sensitive Code Completion", in Proc. ICSME, 2014, pp. 71-80.

[21] M. Pradel , S. Heiniger, T. R. Gross, "Detecting anomalies in the order of equally-typed method arguments", in Proc. ISSTA, 2011, pp. 232-242.

[22] Z. Tu and Z. Su, and P. Devanbu, "On the Localness of Software", in Proc. FSE, 2014, pp. 269-280.

[23] F. Thung, W. Shaowei Wang, D. Lo and J. Lawall, "Automatic recommendation of API methods from feature requests", in Proc. ASE, 2013, pp. 290-300.

[24] "The StackOverflow Question" http://stackoverflow.com/questions/14124328

[25] "The Eclipse JDT Documentation" http://help.eclipse.org/juno/topic/org.eclipse.jdt.doc.isv/reference/api/org/eclipse/jdt/core/dom/Expression.html

[26] G. C. Murphy, M. Kersten, and L. Findlater, "How are java software developers using the eclipse IDE?", IEEE Software, vol. 23, no. 4, 2006, pp. 76-83.

[27] M. P. Robillard, "What Makes APIs Hard to Learn? Answers from Developers", IEEE Software, v.26 n.6, 2009, pp.27-34

[28] "Source code and data" https://asaduzzamanparvez.wordpress.com/parc/

[29] M. P. Robillard , R. Deline, "A field study of API learning obstacles", Empirical Software Engineering, v.16 n.6, 2011, pp.703-732