PARC: Recommending API Methods Parameters

Muhammad Asaduzzaman Chanchal K. Roy Kevin A. Schneider
Department of Computer Science, University of Saskatchewan, Canada
{md.asad, chanchal.roy, kevin.schneider}@usask.ca

***Abstract*—APIs have grown considerably in size. To free developers from remembering every detail of an API, code completion has become an integral part of modern IDEs. Most work on code completion targets completing API method calls and leaves the task of completing method parameters to the developers. However, parameter completion is also a non-trivial task. We present an Eclipse plugin, called *PARC*, that supports automatic completion of API method parameters. The tool is based on the localness property of source code, that is developers tend to put related code fragments close together. *PARC* combines contextual and static type analysis to support a wide range of parameter expression types.**

***Index Terms*—Code completion, API methods, method parameter, Eclipse plugin**

I. Introduction

Modern software development heavily depends on the use of APIs. An API provides access to already implemented functionality to accomplish a particular task. Developers need to learn these APIs to use them effectively during software development. The problem is that APIs have grown considerably these days and that makes them difficult to learn and remember [1]. To free developers from remembering every detail, modern integrated development environments contain a code completion feature. They typically support automatic completion of API method calls but leave the task of completing method parameters to developers. Unfortunately, parameter completion is also a non-trivial task [7]. Unless otherwise specified, we use the term parameter to refer to the actual parameter.

Although some IDEs, such as Eclipse, can suggest method parameters, their support is very limited. For example, consider that a developer is developing a GUI using Java Swing APIs. To add a component to a container, she needs to call its *add* method with two parameters. The first parameter specifies the location on where the component to be added in the container and the second one is the component to be added. While she recalls the method name and the component to be added, she cannot remember what to use to specify the location. She resorts to the parameter completion support of Eclipse. While the method signature tells that the first parameter is of String type but the problem is that there can be a large number of objects whose type matches with the expected parameter type. One solution is to read the API documentation again but that requires additional time and effort. Eclipse fails to recommend the target parameter, which undermines the benefit of code completion and indicates a need for additional support.

In this paper, we present the novel feature, architecture and implementation details of *PARC*, an Eclipse plugin that supports automatic completion of method parameters. The tool collects parameter usage context from past code examples and only considers tokens close to the target parameter position. When a developer requests for a parameter completion, *PARC* matches the current usage context with that of collected examples. Static type analysis is performed next to refine previously matched example parameters to the current development context. The refined parameters are then reported to developers through completion popups. The detail description of the technique including a comprehensive evaluation of the tool can be found else where [8]. The plugin and the source code is available to download from **https://asaduzzamanparvez.wordpress.com/parc/**.

The paper is organized as follows. Sections II describes the features and summarizes the operation of *PARC*. Section III describes the architecture of the plugin. Section IV presents the performance as well as the implementation details. We briefly discuss related work in Section V and Section VI summarizes limitations. Finally, Section VII concludes the paper.

II. Features

PARC is available as an Eclipse plugin and activates automatically when a developer selects an appropriate method call from a completion popup. Since Eclipse Java Development Tools (JDT) supports various different forms of code completions, it is expected that *PARC* does not interfere with code suggestions other than method parameters. To achieve this the plugin intercepts the recommendations made by Eclipse JDT. It also collects current parameter usage context and queries the model with that information. The model contains parameter usage examples from a large number of open source projects. The query returns a list of method parameters and *PARC* puts the top-3 recommended parameters on top of JDT completion proposals by default. However, the number of recommendations made by *PARC* can be configured by the user. The plugin can be enabled or disabled by accessing its own preference page in the Eclipse IDE.

PARC operates in two different phases. Due to space constraints we only summarize these phases briefly as follows (Details of the technique can be found elsewhere [8]):

A. Model Generation

The plugin requires a model for recommending method parameters. A model consists of parameter usage examples collected from open source software systems. A parameter

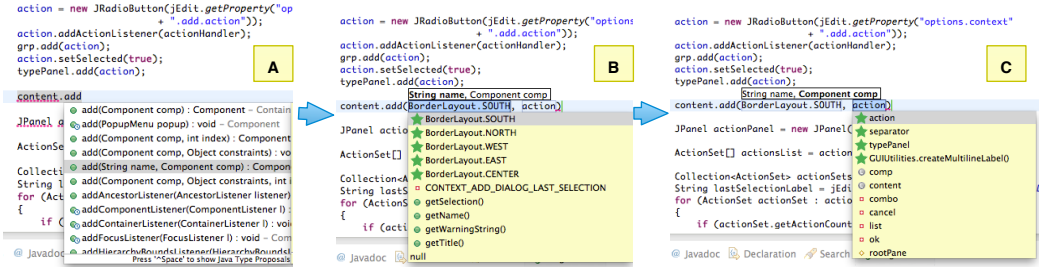

Fig. 1: An example of using PARC to complete method parameters

usage example consists of a parameter usage context, the method name, the receiver type of the method, the type and position of the parameter as described in the method signature. The parameter usage context in our case consists of any method names, any keywords except access specifiers, and any type names that appears within the previous four lines prior to a method parameter position. *PARC* ignores any lines that is either empty or contain only curly braces, comments or a combination of both. The receiver type of the method, its name, and the parameter position are used to index the parameter along with the parameter usage context.

B. Method Parameter Recommendation

This phase consists of the following three steps:

1) Query Information Collection: When a developer selects a name from the method call completion popup, the plugin activates. It collects the current parameter usage context (also referred to as the query context) along with the receiver type of the method call, the name of the method, parameter positions and their types as described in the method signature. The later four pieces of information are used to collect method parameters along with their usage contexts from the model.

2) Determine contextual similarity between query and examples: The next step is to determine the similarity between the query context with that of the examples. The plugin leverages a locality sensitive hashing technique, called *simhash*, to quickly determine a fuzzy distance between the query context and that of examples. It then sorts the associated parameters based on the ascending order of the distance value and selects the top 500 parameters. Next, it sorts the candidate parameters based on the descending order of similarity value calculated using cosine similarity. This is computationally expensive but provides more accurate similarity measure. These sorted parameters are referred to as likely parameter candidates.

3) Static Type Analysis and Recommendation: The objective of this step is to adapt the parameters to the current development context. *PARC* employs rules to do that. For example, if the parameter is a method invocation or a qualified name and the receiver is a simple name, the plugin looks for the variable whose type matches with the receiver type, located within the scope of the method call and is referenced closest to the parameter position. It then replaces the receiver with that variable before making any recommendation.

After removing any duplicates, the plugin recommends the top three parameters by putting them on top of the JDT completion proposals. This number can be changed by setting a different value in the preference page of the plugin.

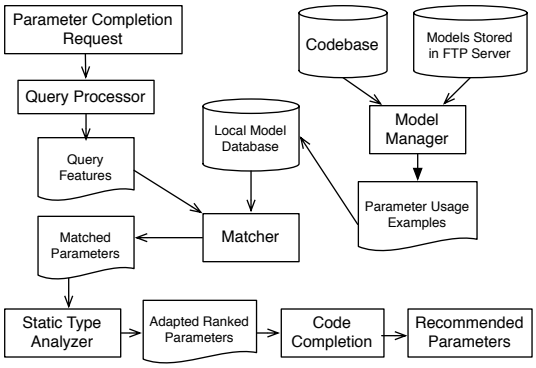

Fig. 2: Architecture overview of PARC

Figure 1 shows an example of the plugin in action. A developer requests for method call completion by pressing a dot after typing the variable name (see Figure 1A). Eclipse shows the possible completion proposals through a popup menu. The developer selects the add method with String and Component parameter types. As soon as the developer makes the selection by pressing the **enter** key, *PARC* activates and recommends completion proposals through another popup menu starting with the first parameter (see Figure 1B). The suggestions made my *PARC* are highlighted with a different icon. The developer can cycle through the other parameters by pressing the **tab** key (see Figure 1C).

III. Architecture

Figure 2 shows the architecture of *PARC*. The plugin consists of four different components. The model manager is responsible for downloading models from a remote server where we plan to periodically update new models. It also generates models from actively loaded projects in the client IDE. The above operations are performed in the background thread to avoid interruption of the current development session. The query processor is responsible for collecting required information for the parameter completion request. The matcher component takes that as input and interacts with the model manager component to look for method parameters whose context matches with the query context. It passes the result to a static type analyzer that adapts the matched parameters with the current development context. Finally, the code completion component accepts the modified ranked method parameters and combines them with JDT completion proposals. If a

proposal is made by both, *PARC* removes the duplicate entry from JDT proposals. Furthermore, the recommendations made by *PARC* appears on top of JDT proposals.

IV. Performance and Implementation

This section briefly summarizes the accuracy and runtime performance of *PARC*. Details of the evaluation procedure and results can be found in a separate paper [8]. We also highlight the plugin implementation.

A. Accuracy and Runtime Performance

We compared *PARC* with the only available state-of-the-art parameter recommendation technique, called Precise. We used code examples from Eclipse and NetBeans software systems. For the first system we used SWT library method parameters and for the second system we used method parameters of Swing+AWT libraries. In both experiments, *PARC* outperformed *Precise*. For example, for the Eclipse system and for the top three recommendations *Precise* achieves 42.68% precision and 30.88% recall values. *PARC* on the contrary achieves 46.29% precision and 43.98% recall value. The difference becomes more significant for the NetBeans system. *PARC* achieves 36.76% more precision and a 33.16% higher recall value than *Precise*. It should be noted that in both experiments we only consider those parameter expressions that are supported by *Precise*.

We also evaluated the execution time of *PARC*. On average, *PARC* required 45 milliseconds to recommend completion proposals which is comparable with other state-of-the-art techniques [7].

B. Implementation

PARC is written entirely in Java. We initially planned to use the *CompletionProposalComputer* extension point of Eclipse to implement the plugin. Neither that extension point nor others contribute to parameter completion proposals. The code responsible for the task is located in the *org.eclipse.jdt.ui* plugin. We update the plugin with additional classes and modify the *ParameterGuessingProposal* class, to combine recommendations of *PARC* with default Eclipse JDT proposals. To put *PARC* recommendations on top of that of JDT, we leverage the relevance value that Eclipse uses internally to sort completion proposals. Thus, the relevance sorting mechanism needs to be selected from the content assist preference page of Eclipse IDE.

V. Related Work

Zhang et al. developed an Eclipse plugin, called *Precise*, that leverages past examples to recommend method parameters [7]. The most notable difference is that *Precise* cannot recommend parameters of the following expression categories: simple name, boolean, null literal and class instance creation. However, *PARC* can recommend parameters of all of the above expression categories including those that are supported by *Precise*. Also both techniques collect the parameter usage context differently. A number of techniques have been developed for code completion, but none focuses on method parameter completion. Bruch et al. developed an Eclipse plugin to recommend method calls using k-nearest neighbour algorithm [3]. Hou and Pletcher developed another Eclipse plugin, called *BCC*, that uses sorting, filtering grouping of APIs to recommend method calls [2]. *CSCC* is a context sensitive method call completion tool [5]. *GraPacc* is a tool that supports automatic completion of API usage patterns [6]. *Calcite* helps developers to instantiate objects of a class [4]. *CACHECA* is a general purpose code suggestion tool based on the cache language model. While the cache component can be integrated into *PARC* to capture locally repetitive parameters, the plugin cannot handle complex parameter expressions.

VI. Limitations

Collecting parameter usage context in *PARC* is simple, requires only tokenization of source code. However, this simplicity comes with the price that tokens that are not related with a method parameter can become part of the usage context. As a result *PARC* sometimes recommends incorrect parameters ahead of the correct one. During our study we observe that some method parameters are only specific to a file or a project. Using a global model can lead to incorrect suggestions in those cases. This matches with the finding of Tu et al. [9]. A possible solution can be interpolating between a local and a global model, and we are currently working on this.

VII. Conclusion

We describe an Eclipse plugin, called *PARC*, that combines the contextual information together with static type information to better recommend method parameters. While the contextual information captures global regularities, static type analysis is incorporated to capture local aspects of the code under development. We are currently working to eliminate the limitations of *PARC* described in the previous section. We also plan to conduct a user study to understand the usefulness of the tool.

References

[1] M. P. Robillard, "What Makes APIs Hard to Learn? Answers from Developers", IEEE Softw., vol. 26, pp. 27-34, Nov. 2009.

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

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

[4] 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.

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

[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] 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.

[8] M. Asaduzzaman, C. K. Roy, K. A. Schneider, "Exploring API Method Parameter Recommendations", accepted to be published in Proc. ICSME, 2015, pp. 10.

[9] Z. Tu, Z. Su, and P. Devanbu, "On the localness of software", In Proc. FSE, 2014, pp. 269-280.