Active Code Completion

Cyrus Omar, YoungSeok Yoon, Thomas D. LaToza, Brad A. Myers
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA, USA
{comar,youngseok,tlatoza,bam}@cs.cmu.edu
https://github.com/cyrus-/graphite

I. Introduction

Software developers today make heavy use of the code completion features available in modern code editors [1]. By navigating and selecting from a floating menu containing the names of variables, fields, methods, types and code snippets, a developer can avoid many common spelling and logic errors, avoid unnecessary keystrokes and explore unfamiliar APIs without leaving the editor window. To ensure that the items featured in this menu are relevant, the editor conducts a static analysis of the surrounding code context.

Several refinements and additions to the code completion menu have been previously suggested in the literature. These have focused on using additional sources of information, such as databases of usage history [2], examples extracted from code repositories [3] and crowdsourced information [4], to increase the relevance and sophistication of the featured items. Empirical evidence presented in these studies suggests that these enhancements further improve developer productivity.

In this paper, we propose a complementary technique called *active code completion*. When the developer invokes the code completion menu, the editor looks for a *palette definition* associated with the type of the expression being entered. If found, an option to use this palette is added to the code completion menu. When the developer selects this option, source code is not inserted immediately. Instead, the palette definition takes control of the code completion interface. The developer can then interact with this interface to provide parameters and other information related to her intent, and receive immediate feedback about the effect these choices will have on the object's behavior. When the developer indicates that she is satisfied with these choices, the palette generates code that is inserted at the cursor.

Before designing and implementing such a system, we sought to address the following questions:

- What are some specific use cases for active code completion in a professional development setting?
- Which functional criteria are common to types that would benefit from an associated palette?
- What usability criteria should inform palette interface designs in this context?
- Which capabilities must the active code completion architecture have to enable these use cases and designs?

To answer these questions, we began by conducting a large online survey of professional developers. Their responses revealed a number of interesting use cases and non-trivial design constraints for active code completion systems.

Based on this information, we designed and implemented an Eclipse plug-in that provides active code completion for the Java language, and implemented some example palettes atop this system. Finally, we evaluated the usefulness of one such palette, designed to assist developers as they write regular expressions, with a controlled lab study. The data and observations from this study provide empirical evidence in support of the view that an active code completion system would be useful to professional developers.

II. Developer Survey

The subject pool for the preliminary survey consisted primarily of visitors to a large programming-oriented collaborative filtering forum[1]. A small number of participants were also recruited using a mass email to local computer science graduate students. Participants were asked to take an online survey taking approximately 20 minutes to complete. Background information collected from the 475 participants indicated that the overwhelming majority of participants were professional developers.

Participants were shown three mockup palettes, along with mockups of the invocation procedure described above, for classes representing colors, regular expressions and SQL queries. They were asked to indicate whether they would find these palettes useful. Their responses are summarized in Fig. 2 and indicate that these developers found the concept potentially useful, with a particular preference for the palette shown for regular expressions.

Participants were also asked to provide free-form comments on each palette. These responses revealed a number of concerns about this system. These include issues related to handling separation of concerns, reinvocation, palette settings and state, interactions between the palette and the code context, language and editor independence, and keyboard navigability. These concerns strongly influenced the design of the code completion system described in the next section.

Finally, we solicited suggestions for classes that may benefit from an associated palette. We received a number of interesting suggestions, which we broadly classified into

[1] http://www.reddit.com/r/programming

978-1-4577-1247-0/11/$26.00 ©2011 IEEE

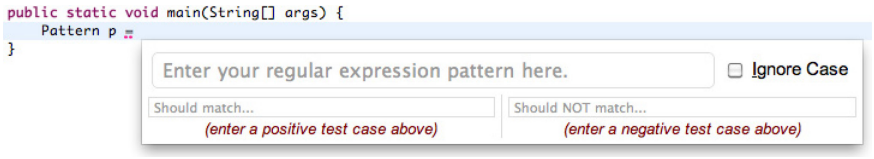

Figure 1. Graphite palette designed for regular expressions

CLASS	Nearly every time	Most of the time	Some of the time	Rarely	Never
Color	9.6%	22.1%	**32.4%**	28.2%	7.7%
RegExp	**36.6%**	29.5%	21.8%	7.3%	4.8%
SQL	18.2%	19.3%	**30.9%**	20.4%	11.4%

Figure 2. The distribution of responses to the question: "Consider situations where you need to instantiate the [specified] class. What portion of the time, in these situations, do you think you would use this feature?

categories. Examples include classes that would benefit from an alternative syntax (e.g. dictionaries), where the implications of a parameter choice are difficult to predict (e.g. 3D transformation matrices), where the instantiation procedure is non-trivial (e.g. factory methods), and where parameters could be provided by example (e.g. key combinations).

III. Prototype Implementation

Next, we implemented an active code completion system as an Eclipse Java plug-in called Graphite, an acronym for **GRA**phical **P**alettes **H**elp **I**nstantiate **T**ypes in the **E**ditor.

A. Association Model

Graphite provides two mechanisms to associate a palette with a class. The *annotation-based association model* allows the developer of a class to associate a palette definition with it using a Java annotation, `GraphitePalette`. The annotation contains the URL of the palette and metadata used to control the description of the palette in the code completion menu. This association model is beneficial because end-users need not realize that a palette is available for a class – when they invoke the code completion menu, they *discover* that a palette exists. This stands in contrast to relevant external tools, which must be explicitly discovered by users.

For classes which are not amenable to direct modification, such as those in the Java standard library, an *external association model* is available in the Graphite preferences pane. Users can explicitly associate a palette with the fully-qualified name of a class using this model.

B. Palette Implementation Model

A common concern expressed in our developer survey was that active code completion should not rely on a particular editor environment. To satisfy this constraint, we did not want to use an IDE-specific user interface framework (e.g. SWT and JFace for Eclipse). Instead, we designed Graphite so that palettes were written using HTML and Javascript. Our preliminary study confirmed that these languages are well-known among Java developers. A small Javascript bridge API was developed to allow palettes to access the current selection in the editor (used to allow reinvocation of palettes) and to insert code at the cursor. The Eclipse plug-in simply displays the palette in a floating browser window, without any associated chrome, and responds to invocations of bridge functions.

This strategy could be straightforwardly replicated in other editor environments, without requiring that palette definitions be modified. Indeed, with appropriate logic in the bridge API, one palette may be able to support several distinct languages.

IV. Lab Study

In order to evaluate the usefulness and usability of Graphite, we conducted a lab study with a palette designed for regular expressions (Fig. 1). Participants were asked to write regular expressions in the Java programming language, in response to several prompts. The control group was not given access to Graphite, but was otherwise free to use online resources. The treatment group was given a short tutorial about the Graphite system using a palette for the Color class as an example. The fact that a palette existed that was relevant to the tasks they would be asked to perform was mentioned, but the palette itself was not explicitly demonstrated, and its use was not required.

Although our present sample size of 7 participants renders quantitative comparison inappropriate, several qualitative observations support the view that Graphite was helpful for users in this task. In particular, we found that the participants in the control group were facing difficulties that the palette was designed to address, including difficulty with the factory pattern used by Java regular expressions and the requirement that backslashes be doubly-escaped. We also observed that few subjects wrote adequate testing code to ensure that their regular expressions were correct. Participants in the treatment group, on the other hand, had few or no difficulties with these issues and wrote tests more frequently, due to the support for testing in the palette interface, and therefore generated more correct code. Feedback from these participants, as well as participants in the control group who were shown the regular expression palette following the experiment, was also positive.

References

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

[2] R. Robbes and M. Lanza, "How program history can improve code completion," in 23rd IEEE/ACM International Conference on Automated Software Engineering, 2008, pp. 317–326.

[3] M. Bruch, M. Monperrus, and M. 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, ser. ESEC/FSE '09. New York, NY, USA: ACM, 2009, p. 213?222, ACM ID: 1595728.

[4] M. Mooty, A. Faulring, J. Stylos, and B. Myers, "Calcite: Completing code completion for constructors using crowds," in Visual Languages and Human-Centric Computing (VL/HCC), 2010 IEEE Symposium on, 2010, pp. 15–22.