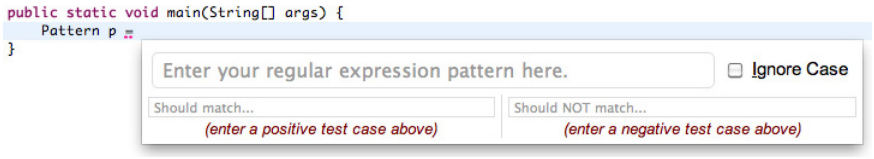

Figure 1. Graphite palette designed for regular expressions

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.