Active Code Completion

Cyrus Omar, YoungSeok Yoon, Thomas D. LaToza, Brad A. Myers
Carnegie Mellon University, Pittsburgh, PA, USA
{*comar,youngseok,tlatoza,bam*}*@cs.cmu.edu*

***Abstract*—Code completion menus have replaced standalone API browsers for most developers because they are more tightly integrated into the development workflow. Refinements to the code completion menu that incorporate additional sources of information have similarly been shown to be valuable, even relative to standalone counterparts offering similar functionality. In this paper, we describe *active code completion*, an architecture that allows library developers to introduce interactive and highly-specialized code generation interfaces, called *palettes*, directly into the editor. Using several empirical methods, we examine the contexts in which such a system could be useful, describe the design constraints governing the system architecture as well as particular code completion interfaces, and design one such system, named Graphite, for the Eclipse Java development environment. Using Graphite, we implement a palette for writing regular expressions as our primary example and conduct a small pilot study. In addition to showing the feasibility of this approach, it provides further evidence in support of the claim that integrating specialized code completion interfaces directly into the editor is valuable to professional developers.**

***Keywords*-code completion, development environments**

I. Introduction

Software developers today make heavy use of the code completion support found in modern source code editors [1]. Most editors provide code completion in the form of a floating menu containing contextually-relevant variables, fields, methods, types and other code snippets. By navigating and selecting from this menu, developers are able to avoid many common spelling and logic errors, eliminate unnecessary keystrokes and explore unfamiliar APIs without incurring the mental overhead associated with switching to an external documentation tool or API browser.

Several refinements and additions to the code completion menu have previously been suggested in the literature. These have focused on leveraging additional sources of information, such as databases of usage history [2][3], inheritance information [3], API-specific information [3][4], partial abbreviations [5], examples extracted from code repositories [6][7] and crowdsourced information [8][9], to increase the relevance and sophistication of the featured menu items. As with the standard form of code completion, many of these sources of data can also be utilized via external tools (e.g. Calcite [8] uses information that could already be accessed using the Jadeite [10] tool). Empirical evidence presented in these studies, however, suggests that directly integrating these kinds of tools into the editor is particularly effective. For example, users of the Calcite tool completed 40% more tasks in a lab study (unfortunately, a Jadeite control group was not included.)

In all of these systems, the code completion interface has remained primarily menu-based. When an item is selected, code is inserted immediately, without further input from the developer. These systems are also difficult to extend: a fixed strategy determines the completions that are available, so library providers cannot directly specify new domain-specific or contextually-relevant logic. In this paper we propose a technique called *active code completion* that eliminates these restrictions[1]. This makes developing and integrating a broad array of highly-specialized developer tools directly into the editor, via the familiar code completion command, significantly simpler. This technique is motivated by the evidence discussed above and further evidence provided in this paper that developers prefer, and make more effective use of, tools that do not require leaving the immediate editing environment.

In this paper, we discuss active code completion in the context of object construction. For example, consider the following Java code fragment:

```
public Color getDefaultColor() {
    return ␣
```

If the developer invokes the code completion command at the indicated cursor position (␣), the editor looks for a *palette definition* associated with the *type* of the expression being entered, which in this case is `Color`. If an associated palette is found, a menu item briefly describing this palette is added to the standard code completion menu. When selected, the corresponding palette is shown, replacing the standard code completion menu. Figure 1a gives an example of a simple palette that may be associated with the `Color` class[2].

The developer can interact with such palettes to provide parameters and other information related to her intent, and receive immediate feedback about the effect these choices will have on the behavior of the object being constructed. When this interaction is complete, the palette generates appropriate source code for insertion at the cursor. Figure 1b shows the inserted code after the user presses ENTER.

[1]Portions of this work previously appeared in a poster abstract [11].

[2]A video demonstrating this process is available at http://www.cs.cmu.edu/~NatProg/graphite.html.

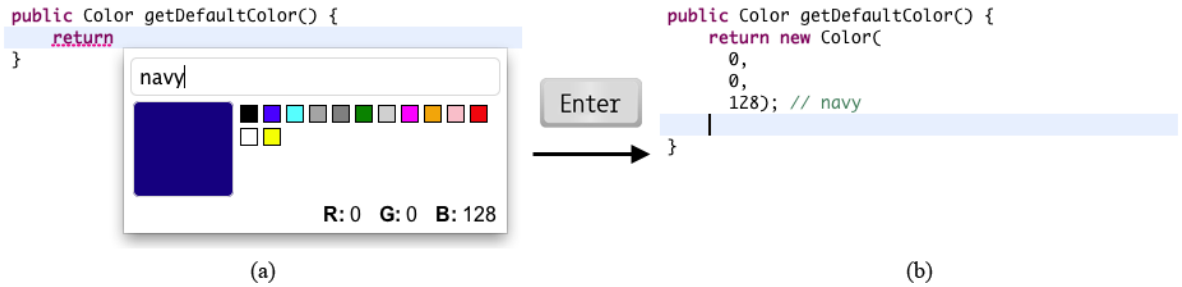

Figure 1. (a) An example code completion palette associated with the `Color` class. (b) The source code generated by this palette.

In accordance with best practices, we sought to address the following questions before designing and implementing our active code completion system:

- What *specific* use cases exist for this form of active code completion in a professional development setting?
- What *general* criteria are common to types that would and would not benefit from an associated palette?
- What are some relevant usability and design criteria for palettes designed to address such use cases?
- What capabilities must the underlying active code completion system provide to enable these use cases and user interface designs?

To help us answer these questions, we conducted a survey of 473 professional developers (Section II). Their responses, along with information gathered from informal interviews and code corpus analyses, revealed a number of non-trivial functional requirements for palette interfaces as well as the underlying active code completion architecture (Section III). Participants also suggested a large number of use cases, demonstrating the broad applicability of this technique. We organize these into several broad categories (Section IV).

Next, we describe Graphite, an Eclipse plug-in that implements the active code completion architecture for the Java programming language (Section V), allowing Java library developers to associate custom palettes with their own classes. We describe several design choices that we made to satisfy the requirements discovered in our preliminary investigations and briefly examine necessary trade-offs.

Finally, we conducted a pilot lab study with a more complex palette, implemented using Graphite, that assists developers as they write regular expressions (Section VI). The study provides specific evidence in support of the broader claim that highly-specialized tools that are integrated directly with the editing environment are particularly useful. We conclude that active code completion systems like Graphite are useful because they make developing, deploying and discovering such tools fundamentally simpler.

II. Survey

To validate our general conceptualization of active code completion, develop concrete criteria to constrain our system and palette designs, and create a list of use cases to justify this effort, we began by conducting a large survey of professional software developers.

A. Participants

We recruited participants for this survey[3] primarily from a popular programming-related discussion forum hosted on the popular website reddit.com [12]. An additional 22 participants were computer science graduate students at CMU.

Recruitment materials in both cases stated that we were seeking developers "familiar with an object-oriented programming language like Java, C# or Visual Basic and an integrated development environment like Eclipse or Visual Studio". Participants were told that the survey would take approximately 20 minutes to complete, and no reward was offered. Of the 696 people who started the survey, 473 participants (68%) completed it. We examine the responses from completed surveys only in the analyses below.

B. Familiarity with Programming Languages and Editors

We first asked participants about their level of familiarity with several programming languages, on a five-point Likert scale[4]. 61.1% of the participants indicated that they were an expert in at least one language, and an additional 35.7% were "very familiar" with at least one language. On average, participants rated themselves as very familiar with Java, C, C++ and JavaScript, familiar with C#, Python and PHP and somewhat familiar with Visual Basic and Perl.

We also asked participants to select which integrated development environments (IDEs) and code editors that they were familiar with. The Eclipse IDE was familiar to 87.1% of participants. This was followed by Visual Studio at 66.0%, Vi/Vim at 53.7%, Netbeans at 37.7%, Emacs at 24.8% and IntelliJ IDEA at 16.4%. Participants could also enter "other" choices and a number of editors and IDEs were entered, including Xcode, Textmate and Notepad++.

[3]https://www.surveymonkey.com/s/2GLZP8V

[4]"None", "Somewhat familiar", "Familiar", "Very familiar", "Expert"

	Regular Expressions	SQL
Separate test script	29.6%	15.4%
Guess and check	14.0%	16.1%
External tool	**37.9%**	**58.6%**
Search for examples	12.3%	5.1%
Other	6.2%	4.9%

Figure 2. Distribution of responses to survey questions asking about typical strategies for writing regular expressions and SQL queries.

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 3. 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?"

C. Palette Mockups

We presented participants with a series of mockup palettes for a Color class (more complex than the one we ultimately implemented in Figure 1a), a regular expression class, and a SQL query class. Participants were also shown mockup screenshots demonstrating how a user could invoke the palette, and a mockup showing the code that would be inserted once a selection had been made. Before presenting each mockup, we gathered information about the strategies that they would normally use to instantiate the class.

For the Color class, the majority of participants indicated that they would look in the code completion menu (58.4%) or in the class documentation (19.0%) for a predefined constant if asked to instantiate an object corresponding to the color "navy" (which is not, in fact, a standard color in Java.) Another 14.0% indicated that they would use an external tool (such as an image editor) to determine the RGB values corresponding to the color.

Before asking participants about regular expressions and SQL queries, we asked participants to rate their familiarity with these concepts. Few participants (4%) indicated that they were unfamiliar with regular expressions and no participants were unfamiliar with creating SQL queries, providing further evidence that our participants were not novice developers. Figure 2 summarizes the strategies that participants generally preferred for instantiating regular expressions and SQL queries. Using an external tool was a common strategy in both cases, particularly for SQL queries, but several other strategies were also represented.

Finally, after showing the series of mockup screenshots, we asked participants to rate how useful the integrated palette would be to them if they needed to instantiate the corresponding class. The responses to this question for each palette are summarized in Figure 3. In each case, more than half of the participants indicated that they would use active code completion at least some of the time. The regular expression palette was considered particularly useful while the color and SQL palettes showed a more reserved pattern of responses.

In addition to asking for a simple rating for each palette, we also solicited open-ended comments. A large number of participants volunteered comments: 193 for the color palette, 129 for the regular expression palette and 142 for the SQL palette. These responses were highly valuable when developing the design criteria below and helped to explain the patterns observed in Figure 3.

III. Design Criteria

Using the information gathered from the survey as well as informal discussions with developers and researchers, we developed design criteria constraining both the overall system design as well as the design of individual palettes. In the section headings below, the number of survey responses, summed over the three palette mockups, that contained the listed concern, as judged by the authors of this paper, are listed in parenthesis. These criteria were useful in designing Graphite (Section V) and we note that this collection of criteria may also be relevant to researchers designing other kinds of editor-integrated tools. Based on the variety of concerns expressed by participants in our survey, the design space for these tools appears to be quite complex.

A. Maintaining Separation of Concerns (183)

The most common issue participants had was that palettes seemed to violate the principle of separation of concerns. The color palette, for example, allows developers to insert color constants directly into the program logic. Many developers noted that this is considered bad practice, or should be limited to the prototyping phase of a project. This concern was also expressed in responses to the SQL palette, which required inserting parameters to connect to a database so that the query could be tested. The resulting code included initialization steps needed to connect to the specific database that was entered, and several participants noted that much of this information should appear in an external resource file separated from the program logic. Few participants made similar comments about the regular expression palette, however, indicating that regular expressions are considered a part of the program logic rather than data by most developers.

This suggests that tool and palette designers may wish to explicitly aprise users of relevant best practices and acknowledge that palettes that generate constant data may be most useful in the prototyping phase. It can be noted that when transitioning from a prototype to production-quality software the code generated by a palette or tool may be used as template to be refactored as needed. It also suggests that resource file and stylesheet editors may particularly benefit from active code completion support.

B. Integration with Testing Frameworks (35)

The regular expression palette shown to the participants allowed users to immediately test a pattern against provided strings. These test strings and the results of performing the match were inserted as comments below the generated source code. A number of participants requested that unit tests be generated instead, likely due to concerns that future modifications might introduce bugs, or due to the desire to conform to standard testing practices. To support the generation of unit tests, the active code completion architecture would need to support code generation at locations other than at the cursor.

C. Support for Reinvocation (19)

Several participants asked for the ability to reinvoke a palette from previously generated source code. In order to support this feature, the architecture must provide the palette with enough information to reconstruct its state. To complicate matters, however, users may wish to modify the generated code between invocations of the palette and have these modifications reflected in the palette's state upon reinvocation (e.g. modify the RGB values in the case of the `Color` class). Moreover, there may be important aspects of the palette's state that are not directly available in the generated code, such as parameters controlling the palette's user interface. Indeed, associating tool-related metadata with code is known to be cumbersome in purely textual languages, since all metadata must be directly visible within comments or annotations.

D. Support for Palette Settings and History (41)

A related feature important to many participants was support for maintaining settings and usage history across invocations of the same palette at different code locations. For example, 20 participants requested that the Color palette include a list of recent or favorite colors, and 12 participants inquired about whether the database connection information was maintained between invocations of the SQL palette.

E. Support for Nested Expressions (13)

In all of the examples that we gave, the parameters entered into the palette interface corresponded to simple, constant expressions, rather than complex expressions referring to variables from the surrounding context. A number of participants noticed this limitation. For example, several participants asked SQL query strings, as these are typically constructed using user-generated data in practice. Although a simple expression entry box may suffice in simple scenarios, architectural support is needed for palettes that need to inspect the code context (e.g. to verify well-formedness) or if code highlighting, code completion and other advanced editing features are needed within the palette itself.

F. Keyboard Navigability (12)

Although our mockup screenshots did not include any completely mouse-driven interfaces, several participants commented that the Color palette included interface elements taken from standard color dialog boxes that could only be manipulated using the mouse. These comments were generally severe in their condemnation of mouse-based interfaces in developer tools, consistent with our finding that a significant portion of our participants were using editors like Vim that place a strong emphasis on keyboard shortcuts.

G. Responsiveness

A common theme in our discussions with developers (as well as in comments left on our recruitment thread) was that integrated development environments like Eclipse were already too slow, and that an extension such as the one we were proposing would only be acceptable if it did not affect performance and responsiveness any further.

H. IDE and Language Portability

The mockups we showed users were based on the Java and the Eclipse IDE. As we showed, a number of participants preferred other languages or editors. Many of these participants made comments asking that the features we describe be IDE and programming language independent. Indeed, the palettes we demonstrated could be used with only slight modifications in a variety of programming languages, given suitable architectural support for porting palettes between editing environments.

I. Varying User Needs

While some participants wanted simpler palettes, others requested significant new capabilities, indicating that user needs may vary substantially. For example, our initial color palette (different from the one shown in Figure 1) was deemed overly complex by many participants (26). Other users wanted additional features, such as an "eyedropper" tool (11) for selecting a color directly from an image on the screen. The regular expression and SQL palettes were considered too simple (by 12, and 15 participants respectively). Users suggested syntax highlighting, several mechanisms for generating, testing and sharing regular expressions and queries, and the ability to browse SQL databases.

Due to this wide range of user needs even for a single class, it may be that support for multiple palettes or a tabbed palette interface would be helpful in practice. To support incremental improvements based on such user feedback, an architecture that makes deploying new and updated palettes relatively painless would also be valuable.

IV. Use Cases

At the end of the survey, we asked the the participants to suggest other classes that could benefit from an associated palette to support our claim that active code completion

is broadly applicable, and to allow us to characterize the specific scenarios where it may be most useful. A total of 119 participants made one or more suggestions, which we classified into several broad categories (we omit a few of these below due to space constraints). As above, the number of participants suggesting a palette in each category is listed in parenthesis. We also include suggestions made by researchers and developers in private discussions without including them in the provided counts.

A. Graphical Elements (27)

The most popular suggestions were graphical elements, influenced perhaps by our demonstration of the Color palette. Some participants suggested palettes for classes representing primitive graphical objects, such as brush and font selectors or polygon editors, while other participants were focused on user interface elements, such as buttons, check boxes and frame layouts. A few also suggested palettes for manipulating 3D primitives, such as transformation matrices, in a more direct and intuitive manner. A practitioner also suggested that because setting up a plot or graph is often significantly simpler using a direct manipulation interface, it would be a natural candidate for a palette as well.

B. Query Languages (17)

The second most popular category of suggestions consisted of various interfaces for query languages, also likely due to the examples we provided to participants. In addition to variants of the SQL and regular expression palettes, developers also wanted to work with other types of queries such as XPath or XQuery for XML.

C. Simplified or Domain-Specific Syntax (16)

Another interesting class of suggestions were cases where a more natural syntax than the syntax provided by Java is desirable. One suggestion was a palette that automatically escaped strings containing quotation marks or escape sequences. A related category of suggestions consisted of palettes that offered a more natural interface for generating strings containing code in other languages such as HTML (e.g. offering syntax highlighting, escaping, tag matching and other features.) Domain-specific syntax for complex mathematical expressions and chemical formula were also mentioned in discussions with practitioners.

An interesting suggestion that we investigated further involved Java's collection classes, such as `ArrayList` and `HashMap`. A participant suggested that these classes could be associated with a palette that offered a simplified literal syntax for initialization, pointing toward other languages that do offer such a literal syntax (e.g. JavaScript.) Without such syntax, these classes must be tediously initialized using a separate method call for each element. To determine whether this usage pattern is common, we conducted a corpus analysis using 10 randomly selected projects from the Qualitas Corpus [13] containing over 1M lines of code. We began by searching for places in these projects where Java collection classes were being instantiated, then looked to see whether this instantiation code was immediately followed by method calls that inserted items into the collection, indicating a case where a literal may have been used if available. Figure 4 summarizes the results of this analysis, providing evidence in support of the claim that a palette that simplifies this process could be useful for general-purpose programming.

Collection Class	Total	Literal	Percentage
`ArrayList`	464	44	9.5%
`HashMap`	56	19	33.9%
`HashSet`	122	62	50.8%
`Hashtable`	86	10	11.6%
`Vector`	729	31	4.2%
Total	1457	166	11.4%

Figure 4. Usage patterns for common Java collection classes in the `java.util` package in our code corpus. Uses that fit a pattern that can be captured by a literal make up a significant portion of all uses. Not all possible usage scenarios of this type were captured by our analysis, so these numbers are lower bounds.

D. Unclear Parameter Implications (11)

Another category of use cases contains classes where it can be difficult to predict what the run-time behavior of a particular parameter choice may be. Examples given included audio filters (e.g. pitch manipulation) and animation descriptors (e.g. speed or shape parameters). By giving immediate visual or auditory feedback using a preview panel, these parameters can be tweaked without requiring the execution of the full application.

E. Integrating with Documentation and Examples (7)

Some participants suggested integrating tutorials or lists of relevant examples directly using a palette, so that these can be discovered more easily by new users and inserted directly into code, without requiring switching to a web browser and executing a search.

F. Complex Instantiation and Cleanup Procedures (5)

A related category contains classes that require complex instantiation and cleanup procedures. For example, in order to read a text file in Java, the developer might want to use `BufferedReader` class. This class can be difficult to use because it requires try/catch block, and one must remember to close the file after reading it. By using a palette to choose a file or choose a variable which contains the file path, the developers could easily instantiate these objects and get an outline containing the full life-cycle of the file. Similarly, palettes may help to alleviate the factory pattern usability problem [14]. As long as the developers remember which class to use, they will not need to remember how to instantiate that class. We explore this further in our user study in Section VI.

G. Instantiation by Example (2)

In some cases, it is possible to describe an object by example. For instance, a class that represents a shortcut key combination may be most easily instantiated using a palette that simply reads a shortcut key from the developer.

H. Proof Assistants

A proof assistant is a tool for constructing proof terms. According to the well-known Curry-Howard isomorphism between programming languages and formal logics, proof terms correspond to expressions and propositions correspond to types [15]. Active code completion works directly with types to help developers construct expressions, so if applied to a language with a cleanly-developed connection to formal logic (e.g. Coq), palettes would be useful for constructing interactive proof assistant interfaces.

V. System Design and Implementation

After completing the survey, we built an active code completion system named Graphite, an acronym for **Gra**phical **P**alettes to **H**elp **I**nstantiate **T**ypes in the **E**ditor. We chose to build the system as an extension to Eclipse for Java because this combination was the most widely-used amongst participants in our survey. In the subsections below, we describe how several novel design decisions made it possible to satisfy design criteria from Section III and enabled several use cases described in Section IV. The end result is a simple system that allows an API's developers, as well as external developers, to build rich HTML5-based palettes that can be associated with both in-built and user-defined classes directly. Developers using an API can discover and invoke these palettes through the standard code completion menu.

A. HTML5-Based Palettes

Palette developers build palettes using HTML5 technologies (HTML, CSS and JavaScript). We made this decision for several reasons. Eclipse is written in Java and uses the SWT and JFace graphical user interface toolkits, but these are not widely used outside of the Eclipse ecosystem. JavaScript was among the most well-known languages in our study, just behind Java, C and C++, and is highly flexible. A number of useful libraries are available (e.g. jQuery [16].)

As noted in Section IV-H, a number of participants in our field study indicated that they hoped that our tool would be available for other IDEs and other programming languages. All major windowing toolkits feature a web browser control, so HTML5-based palettes can easily be loaded by different editor environments without developer intervention.

Deploying palettes using standard URLs is also simpler than attempting to integrate them directly into Java libraries and packages. It also eases the process of incremental and rapid development, as all major browsers now feature sophisticated debuggers and run-time inspection facilities.

B. Palette API

Palettes communicate with the host IDE using a simple Javascript API. To access this API, the palette must include a small script named `graphite.js` into their page. The default implementation of this script provides an implementation of the API methods for testing in a standard web browser. When loaded into an editor, the editor plug-in replaces these methods with specialized implementations. The API consists of the following methods, accessed through a global object named `graphite`:

- `insert(str)`: Inserts the specified string at the cursor and closes the palette. The indentation is automatically inserted after any newlines in this string.
- `cancel()`: Closes the palette without inserting code. Note that the ESC key cancels by default, and palettes are modal so clicking outside a palette will not cancel.
- `getSelectedText()`: Returns the text that is currently selected in the editor, or an empty string if no text has been selected. This method is used to implement reinvocation – users simply select previously generated code and invoke the palette as described in Section I. The palette is responsible for parsing the selected text to extract relevant parameters.
- `getIDE()`, `getLanguage()`: Return a string that specifies the IDE and language that is being used, "Eclipse" and "Java" in our implementation.

By limiting the complexity of this API, we reasoned that developers would be able to create specialized palettes more easily. A simple hello world palette is only two lines of code, for example. An additional benefit is that the developers of other editing environments should also be able to create plug-ins that support Graphite palettes with minimal effort.

C. Palette Discovery

Graphite currently provides two methods for associating a palette with a particular class so that the editor plug-in can include it in the code completion menu when relevant.

1) Annotation-based: For user-defined classes that the palette developer has the authority to modify, the `@GraphitePalette` annotation associates a palette with the class. The annotation must specify the URL of the palette, and can contain some other optional information (e.g. the description that is shown in the code completion menu). This allows API providers to provide palettes that are specialized to their libraries and distribute them directly alongside their code. The benefit of this approach is that users of the API are not required to discover that an external tool exists and explicitly install it into their IDE.

2) Explicit: In cases where a palette developer cannot modify a class directly (such as palettes for classes in the Java standard library), end-users can explicitly associate a palette with the fully qualified name of a class via a preference pane in the Eclipse IDE.

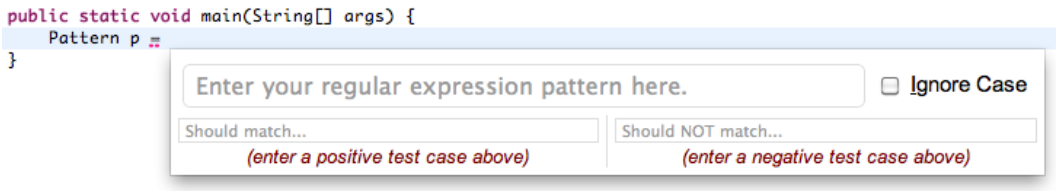

Figure 5. The regular expression palette developed using Graphite and used by subjects in the treatment group of the user study described in Sec. VI.

D. Design Trade-Offs

The design that we have described is light-weight, highly flexible and does not significantly impact IDE responsiveness. It also lays foundations for IDE and language portability. However, this design also leads to trade-offs:

- Because palettes are implemented in Javascript, any differences between the semantics of Java and Javascript can be problematic. For example, color names are slightly different between Java and Javascript, as are the regular expression engines. Although a Java applet could be used in cases where these differences are critical, this is still more difficult than it would be if the palettes were implemented in the same language.
- The palette user interface must stay within its bounding box – user interface elements like pop-up menus (unless they are provided by the browser itself) are thus more difficult to implement and may require additional API support in the future.
- Several use cases could benefit from greater access to the surrounding code, or even the surrounding project. Implementing this modularly is particularly difficult, and the Eclipse API does not easily allow for serialization of the sum of its contextual knowledge for consumption by a palette.
- Because the reinvocation mechanism relies on parsing the selected code, the burden is high for both palette developers (although Javascript libraries for parsing Java code are available) and palette users. A better solution would be one where palette-related metadata is stored directly with the generated code. This could be partially addressed by the use of special comments to delimit sections of generated code and store palette metadata, but a more elegant solution would require a change in how Java source code is represented.

E. Palettes

We implemented two palettes using the jQuery library for basic functionality [16], which we used in the pilot study described below.

1) Color Selection: The color selection palette we ultimately developed was significantly simpler than the one shown in the preliminary survey, due to user comments. It allows users to enter any valid CSS color string, satisfying the requirements for keyboard navigability. The entered color is syntax checked and a preview is shown. Standard Java colors are also available as swatches that can be selected by the mouse if desired. Reinvocation support is provided. This palette took about 500 lines of code and markup, not including the jQuery library. Much of this was needed to translate arbitrary CSS color strings into RGB values, rather than for user interface logic.

2) Regular Expressions: Writing correct regular expression patterns is difficult. The regular expression palette, associated with the `Pattern` class in the Java standard library, allows users to enter and test regular expression patterns interactively before inserting them into their code. The focal point of this palette is the pattern input area. As the user enters a pattern into this input area, syntax errors are indicated with a red background; the background remains white when the pattern is valid.

In addition to the pattern, a regular expression consists of flags that can change its matching behavior in various ways. Our palette allows users to toggle the case-sensitivity flag of the regular expression using a small checkbox labeled `Ignore Case`, placed next to the input area. A keyboard shortcut is also available, indicated using the underlined letter (Ctrl+I).

To allow users to test the behavior of the regular expression that they have entered, two columns are available below the expression input area. The left column contains an input area labeled “Should match...” and the right column contains an input area labeled “Should NOT match...”. Users use these to enter lists of test strings into each column. The background colors behind these strings change to indicate whether the regular expression that has been entered matches or does not match that string. Green indicates that the pattern *matched* the string and red indicates that it does *not* match, regardless of the column that the test is in (this scheme was chosen based on feedback from an initial pilot of this palette, the results of which are not included below.) A key describing this color scheme is displayed after the first test has been entered (not shown above, see video).

Users can navigate the palette using the keyboard using standard `Tab` cycling behavior. The label in each text area remains visible until some input has been entered, rather than disappearing immediately on focus. When a user is satisfied with the regular expression that they have entered,

she can press the `Enter` key to insert the appropriate Java source code. Because Java requires that the regular expression pattern be placed inside a string literal, additional escape sequences are needed in front of backslashes. This can be tedious and error-prone if done manually. The palette automatically inserts these escape sequences. In addition to the source code itself, the tests are retained in a comment beginning on the next line. If the user wishes to modify the regular expression or change the test set, she can highlight the code that was inserted and then invoke the palette once again. The palette parses the selected text to extract the regular expression and tests.

This palette required about 700 lines of code and markup.

VI. Pilot Study

We conducted a small controlled pilot study to evaluate the usefulness and usability of the Graphite system for a specific development task – writing regular expressions – and found significant benefits for the treatment group.

A. Study Methods

1) Between-Subjects Design: We used a between-subjects design by randomly assigning the participants to either the control group or the treatment group. In the control group, subjects were not shown or able to use any palettes. In the treatment group, subjects were shown the simple color palette shown in Figure 1, and allowed to discover and use the regular expression palette shown in Figure 5 if they wished. No specific training on the use of this palette was provided, to simulate realistic usage scenarios.

We chose a *between-subjects design* because a within-subjects design would have required us to produce pairs of tasks with equal difficulties. This turned out to be very challenging. Second, we could not easily ignore a learning effect during the experiment if we had used a within-subjects design. We observed that this effect was quite strong, as most subjects had not used regular expressions recently.

2) Training: Only the participants in the treatment group were shown how to invoke Graphite palettes in the context of an Eclipse code editor with a palette for the Color class. We chose to demonstrate the tool using a color palette instead of the regular expression palette itself because we wanted to simulate the condition where a user had discovered the palette naturally. The demonstration of the Color palette was brief, taking about two minutes. We then described the nature of the task to the subject and allowed them to begin, giving them 45 minutes to complete all tasks, in any order.

3) Tasks: There were a total of 9 tasks to be completed in 45 minutes. The first 6 tasks involved writing regular expressions to validate various data formats (e.g. temperatures), and the remaining 3 tasks involved writing regular expressions to retrieve data from a document. The participants were allowed to move back and forth while doing the experiment, and they were allowed to use any external resources, including the internet, local console, and so forth (we did not observe any usage of programs other than the web browser, however). The only restriction we placed on their activity was that they were NOT allowed to directly search for the answer to a task online. We omit descriptions of each individual question due to space limitations.

B. Participants

We recruited 7 PhD students from CMU. The subjects were randomly assigned into two groups: 4 subjects were assigned to the control group, and the other 3 subjects to the treatment group. There were six male participants and one female participant. Participants were compensated in the amount of 15 dollars for their participation.

All subjects had prior experience with both Java and regular expressions, assessed using a preliminary survey similar to the one described in Section II. Of note, the subjects in this study were slightly less skilled with Java and regular expressions than in the previous survey. This is consistent with the responses of the PhD students who participated in the online survey, who were also slightly less experienced on average. The only other significant difference in responses was that the PhD students were slightly less likely to prefer the use of external tools. The difference in the case of regular expressions was 13%.

C. Hypotheses

In designing the regular expression palette, we had hypothesized that users would experience difficulties with two particular aspects of the Java Pattern API: that it used a factory pattern for instantiation, as opposed to the standard instantiation construct [14], and that it required special care with escape sequences – escape sequences in the pattern must themselves be escaped, because the regular expression is written as a Java string literal. We observed difficulties with both of these issues in the control group, but not in the treatment group. The treatment group also completed more tasks than the control group on average (7 vs. 6.) The behaviors observed in these groups are described below.

D. Preliminary Reading

All subjects were somewhat rusty on the details of the Java Pattern API. After reading the prompt for the first task, all subjects began by searching for and reviewing documentation related to regular expressions. In all but one of these cases, the subject looked at the API documentation for the Java Pattern class during this initial review. The remaining subject, a member of the control group, referred to quick reference documentation provided by an external tool. In all cases, the documentation was left accessible in a browser window throughout the study, often displayed simultaneously alongside the coding window due to the large screen available for the study.

E. Control Group

In our previous online survey, we had found that no single strategy for writing regular expressions dominated the others. We observed each of the common strategies – use of external tools, test scripts and guess-and-check.

One subject began by using an external tool called regexpal.com. After writing a full regular expression and attempting to test it using the tool, the subject became dissatisfied with it and switched to a different tool, regextester.com. This tool too appeared to be unsatisfactory, as all subsequent tasks were performed without the aid of external tools or tests.

The other subjects chose to write their regular expressions directly within Eclipse from the beginning. One of these subjects never compiled or checked the accuracy of the regular expressions beyond making sure Eclipse errors were addressed, while the other two attempted to write Java stubs within the test files to test the regular expression that they had written. They experienced considerable difficulty with this task, with one subject taking over 10 minutes to write the test code. The code could only check one example at a time and the subject used it to check a single positive example per task.

One of the four control subjects tried the `new Pattern` construct. An Eclipse error alerted the subject to the problem shortly thereafter. After referring to an example in the inline API documentation for the class, the subject was able to correct this error. The remaining subjects all noticed this example beforehand, and were able to avoid this problem.

Three of the four control subjects experienced significant difficulties associated with escape sequences. One subject recognized the error fairly quickly after Eclipse complained that the escape sequences used in the pattern were invalid (they were, in fact, valid escape sequences for regular expression patterns, but not for string literals.) The subject continued to miss escape sequences occasionally throughout the remainder of the study, but was able to fix them quickly after Eclipse alerted the subject to the error. For the other two subjects, the problem was more severe. In both cases, they noticed the error that Eclipse gave, but thought it indicated a problem with their pattern itself. One subject thought that the uses of symbolic escape sequences like `\(` were incorrect. He decided to replace these with ASCII escape sequences (`\050`) after looking up an ASCII conversion table. This was an unnecessary and overly complex solution to the problem. The other subject thought that the problem was in his use of the whitespace escape sequence, `\s`. Thus, he spent several minutes looking up how to match whitespace correctly. In doing so, he found a description of the double-escape problem in Java and fixed the problem after that.

F. Treatment Group

One subject was already familiar with an external tool, regexpal.com (also used by a subject in the control group), and used it variously throughout the task. In most cases, the subject used the external tool's quick reference documentation while using our palette for authoring and testing. At other times, the subject used the external tool itself. The subject indicated that in some cases, he was not sure that our palette was free of bugs (when something unexpectedly matched or did not match. We did not observe any actual errors related to regex matching in our tool, however.), and also because the external tool provided syntax and substring highlighting, a useful feature for complex regular expressions. When done with the external tool, the subject pasted the pattern into our tool to generate the appropriate code. As such, the subject had no difficulties with escaping or factory pattern instantiation.

The other two subjects in the treatment group did not use any external tools. Both subjects decided not to use our palette for the first task (likely because they forgot that it was available, since they spent some time looking up API documentation after our initial demo with the color palette.) They both recognized the need for the factory pattern and also both had an initial error related to escaping that they were able to resolve before moving on. One of these subjects also began to write tests within a `main` method.

Beginning with the second task, both subjects remembered that a palette may be available and were able to invoke it correctly. They all recognized that the primary input box was where the regular expression should be entered and that the two other input boxes were for positive and negative tests, respectively. Two of the subjects did not initially realize that multiple tests could be entered, and that entering even a single test required pressing the `Enter` key. One subject never fully understood this, relating after the study that he thought that the palette would notify him when he was done if the example that he had entered was not matched by the pattern. The other subject was able to use the test case input boxes correctly after initial experimentation. This was perhaps due to the wording of the prompts under these entry boxes: "enter **a** [positive | negative] test case above". This problem was corrected in a version of the palette developed following this study.

Two of the subjects used the reinvocation feature of the palette, but neither highlighted the test cases, meaning that they had to enter new test cases every time they reinvoked the palette. This may have been due to the fact that our example with the Color palette involved only a single line of text (unlike the example in Figure 1). The third subject never used this feature.

Two of the subjects expressed confusion about the meaning of the green and red backgrounds on the test cases. Although we provided a key saying that green meant that "pattern matches string", the headings of the two columns: "[should | should NOT] match:", may have caused confusion. We had changed the meaning of these colors due to feedback from our initial presentation of the tool, so

it is clear that regardless of the interpretation given, some subjects were confused.

Despite these difficulties, however, the basic functionality of the palette seemed to help all of the subjects. None struggled with issues related to escaping and instantiation, for example. Although the sample size is not large enough to make many quantitative judgements, it was observed that the treatment group completed more of the tasks than the test group on average (7 tasks for the treatment group vs. 6 for the control group). All of the subjects in the treatment group, as well as members of the control group who were shown the palette after the study, indicated that they felt that the palette was helpful, and several provided specific suggestions for improvements.

G. Threats to Validity

In addition to the small sample size and the sampling bias discussed earlier, subjects in the treatment group may have been biased to use our features due to their novelty (although two subjects did not use the palette until their second task). We only tested a regular expression palette, so other types of palettes may not necessarily be as useful, and this palette had significant flaws that were corrected only after this study.

VII. Related Work

In addition to the code completion work discussed in the introduction, some other research areas are related to active code completion.

A. Active Libraries

We named this technique *active code completion* because of its relation to the general concept of *active libraries* [17]. Active libraries are libraries that contain program logic that is invoked at either compile-time or, here, design-time.

B. Visual Languages

Because active code completion involves graphical user interface elements but ultimately generates textual source code representations, it can be considered a hybrid approach that borrows interaction techniques from visual languages while remaining compatible with conventional programming languages. This hybrid approach may help address some of the usability challenges previously associated with visual languages (cf. [18]).

Editing environments like Barista [19] and the RBA editor [20] also merge concepts from both text-based and structured editors by allowing for alternative code representations within a relatively conventional layout. Barista provides the opportunity for rich type-specific interfaces, but it is an IDE generation framework, so new extensions require recompilation. The RBA editor focuses on code readability rather than new modes of interaction, but new registrations can be added relatively easily. Both tools use a custom domain-specific language, which is likely to be unfamiliar to many users.

In future work, we hope to explore an extensible, keyboard-driven, code-generation based approach that leverages structured code representations to eliminate the difficulties associated with reinvocation and maintaining palette state described in this paper.

C. Specific IDE Features

There exist some IDE features that have been specifically designed for certain types. For example, CodeRush [21] and Resharper [22] have color dialogs that allow developers to launch a color picker directly from the code editor. IntelliJ IDEA has an inline regular expression palette, driven by its Intentions system, as well [23]. However, these IDE specific features are hard-coded – user-defined types cannot provide similar functionality. Recent versions of Visual Studio support user-defined palettes associated with specific fields, rather than classes, of user interface widgets [24]. These are shown only in the property pane when using the graphical window layout editor.

VIII. Conclusion

Motivated by evidence that integrating highly-specialized tools directly into a developer's workflow is useful, we have developed the concept of active code completion as a generalization of conventional code completion. We validated the usefulness by generating a number of use cases and developed general design constraints for such tools, as well as the underlying architecture, by conducting an extensive survey of professional developers. Based on these findings, we developed Graphite, an active code completion architecture that makes several novel design decisions that ease the development, deployment and discovery of user-defined palettes. We created palettes for color and regular expression classes and validated the latter palette's usefulness with a pilot study, providing evidence for the more general claim that integrating palettes into code completion is useful. We claim that active code completion systems like Graphite will considerably ease this process.

IX. Availability

Graphite is free, open-source software available from the URL on the first page. We encourage readers to install it and tell us about any interesting palettes that they develop.

Acknowledgment

We would like to thank the participants in our survey and pilot study, Jonathan Aldrich, the PLAID group, the students of 05-899D, anonymous reviewers and the UIUC Software Engineering Seminar group for valuable feedback. This material is based upon work supported by the National Science Foundation under Grant No. CCF-0811610. CO is supported by a DOE CSGF under Grant No. DE-FG02-97ER25308 and NSF grant CCF-1116907, and YY is supported by the Korea Foundation for Advanced Studies.

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 *Proc. 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE'08)*, 2008, pp. 317–326.

[3] D. Hou and D. Pletcher, "An evaluation of the strategies of sorting, filtering, and grouping api methods for code completion," in *Proc. 27th IEEE International Conference on Software Maintenance (ICSM'11)*, 2011, pp. 233 –242.

[4] H. M. Lee, M. Antkiewicz, and K. Czarnecki, "Towards a generic infrastructure for framework-specific integrated development environment extensions," in *Proc. 2nd International Workshop on Domain-Specific Program Development (DSPD'08), co-located with OOPSLA'08*, 2008.

[5] S. Han, D. R. Wallace, and R. C. Miller, "Code completion from abbreviated input," in *Proc. 2009 IEEE/ACM International Conference on Automated Software Engineering (ASE'09)*, 2009, pp. 332–343.

[6] M. Bruch, M. Monperrus, and M. Mezini, "Learning from examples to improve code completion systems," in *Proc. 7th European Software Engineering Conference & ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'09)*, 2009, pp. 213–222.

[7] J. Brandt, M. Dontcheva, M. Weskamp, and S. R. Klemmer, "Example-centric programming: integrating web search into the development environment," in *Proc. 28th ACM Conference on Human Factors in Computing Systems (CHI'10)*, 2010, pp. 513–522.

[8] M. Mooty, A. Faulring, J. Stylos, and B. Myers, "Calcite: Completing code completion for constructors using crowds," in *Proc. 2010 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'10)*, 2010, pp. 15–22.

[9] "Snipmatch." [Online]. Available: http://languageinterfaces.com/

[10] J. Stylos, A. Faulring, Z. Yang, and B. A. Myers, "Improving API documentation using API usage information," in *Proc. 2009 IEEE Symposium on Visual Language and Human-Centric Computing (VL/HCC'09)*, 2009, pp. 119–126.

[11] C. Omar, Y. Yoon, T. LaToza, and B. Myers, "Active code completion," in *Proc. 2011 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'11)*, 2011, pp. 261–262.

[12] "reddit - programming." [Online]. Available: http://www.reddit.com/r/programming

[13] E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble, "Qualitas corpus: A curated collection of java code for empirical studies," in *Proc. 2010 Asia Pacific Software Engineering Conference (APSEC'10)*, 2010.

[14] B. Ellis, J. Stylos, and B. Myers, "The factory pattern in API design: A usability evaluation," in *Proc. 29th International Conference on Software Engineering (ICSE'07)*, 2007, pp. 302–312.

[15] B. C. Pierce, *Types and Programming Languages*. MIT Press, 2002.

[16] "jquery: The write less, do more, javascript library." [Online]. Available: http://jquery.com/

[17] T. L. Veldhuizen and D. Gannon, "Active libraries: Rethinking the roles of compilers and libraries," in *Proc. 1998 SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing*, 1998. [Online]. Available: http://arxiv.org/abs/math/9810022

[18] P. Miller, J. Pane, G. Meter, and S. Vorthmann, "Evolution of novice programming environments: The structure editors of carnegie mellon university," *Interactive Learning Environments*, vol. 4, no. 2, pp. 140–158, 1994.

[19] Ko, A. J., Myers, and B. A., "Barista: An implementation framework for enabling new tools, interaction techniques and views in code editors," in *Proc. ACM Conference on Human Factors in Computing Systems (CHI'06)*, 2006, pp. 387–396.

[20] S. Davis and G. Kiczales, "Registration-based language abstractions," in *Proc. ACM international conference on Object oriented programming systems languages and applications (OOPSLA'10)*, 2010, pp. 754–773.

[21] "Show color - online documentation - developer express inc." [Online]. Available: http://documentation.devexpress.com/#CodeRush/CustomDocument8887

[22] "Color assistance." [Online]. Available: http://www.jetbrains.com/resharper/webhelp/Coding_Assistance__Color_Assistance.html

[23] "How to check your regexps in intellij idea 11?" [Online]. Available: http://blogs.jetbrains.com/idea/tag/regexp/

[24] "Custom design-time control features in visual studio .net." [Online]. Available: http://msdn.microsoft.com/en-us/magazine/cc164048.aspx