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
http://www.cs.cmu.edu/~NatProg/graphite.html

***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*, a technique that enables the integration of a much broader array of interactive and highly-specialized code generation interfaces, called *palettes*, directly into the editor environment. 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 controlled user study that provides further evidence in support of the claim that integrating specialized tools directly into the editor environment is both feasible, particularly if aided by an active code completion system like Graphite, and 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] and crowdsourced information [7], 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 [7] uses information that could already be accessed using the Jadeite [8] tool). Empirical evidence presented in these studies, however, indicates that directly integrating these kinds of tools directly into the editor environment can be more effective.

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 limited in that they are difficult to modularly extend. A fixed strategy determines the completions that are available, leaving library providers unable to specify new domain-specific or contextually-relevant items. In this paper we propose a technique called *active code completion* that eliminates these restrictions. 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.

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 is responsible for generating appropriate source code for insertion at the cursor. Figure 1b shows the code that is inserted after the user presses the ENTER key.

In accordance with rational design practices, we sought to address the following questions before designing and

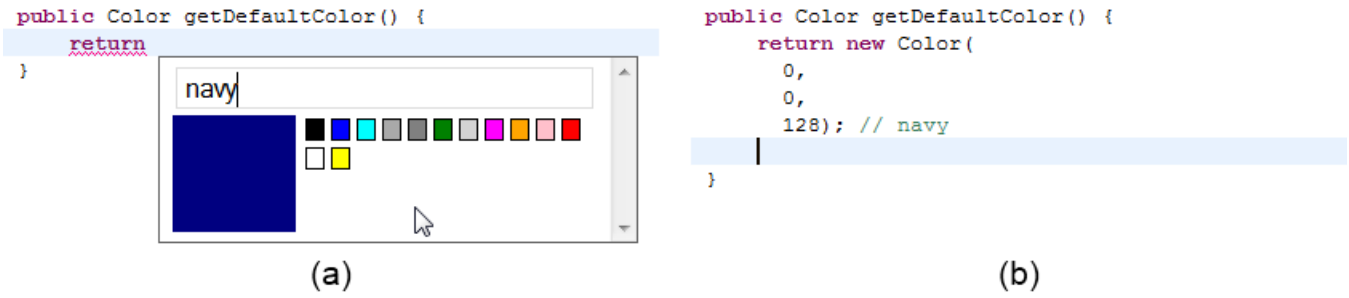

Figure 1. (a) An example code completion palette associated with the `Color` class. (b) The source code generated by this palette.

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 large online survey of professional developers (Section 2). Their responses, along with information gathered from informal interviews and code corpus analyses, revealed a number of interesting use cases (Section 3) and non-trivial functional requirements for palette interfaces as well as the underlying active code completion architecture (Section 4). Notably, many of these findings can be applied to development tools for professional developers more generally, not simply those integrated into the editor in the way we have shown.

Next, we describe Graphite, an Eclipse plug-in that provides active code completion for users of the Java programming language (Section 5). We describe several interesting design choices that were made to satisfy many of the requirements discovered in our preliminary investigations and briefly examine the trade-offs that were necessary due to Java's purely textual program representations and Eclipse's limited extension APIs.

Finally, we conduct a controlled lab study with a more complex palette, implemented using Graphite, that assists developers as they write regular expressions (Section 6). The data and observations gathered from this study provide specific evidence in support of the broader claim that highly-specialized tools that are integrated directly with the editing environment are particularly useful, and that an active code completion system like Graphite makes developing, deploying and discovering such tools fundamentally simpler.

II. Survey Results

To validate our general conceptualization of active code completion, develop concrete design criteria and create a concrete list of use cases, we began by conducting a large survey of professional software developers.

A. Survey Methods

We recruit participants for this survey[1] primarily from a popular programming-related discussion forum hosted on the popular website reddit.com[2]. An additional 22 participants were recruited from the local computer science graduate student body.

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 ranging from "None" to "Expert". 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++.

[1]https://www.surveymonkey.com/s/2GLZP8V
[2]http://www.reddit.com/r/programming

	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

Next, 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. Fig. 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 Fig. 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 Fig. 3.

III. Design Criteria

Using the information gathered from the survey as well as informal discussions with developers and researchers over the course of this study, we sought to develop 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. Many of these concerns apply generally to various kinds of editor-integrated tools, not just those implemented using the active code completion mechanism that we focus on, so we hope these design criteria will be valuable to the developer tool design community more broadly.

A. Maintaining Separation of Concerns (183)

The most common issue developers had, particularly with the Color palette, was that it seemed to ignore the issue of separation of concerns by allowing 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 code that was inserted 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 rather than inline with the program logic itself. 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 feedback 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 that can be refactored to follow stricter policies as needed. We showed users the Color and SQL palettes in the context of a standard programming editor, but these

comments indicate that resource file and stylesheet editors may benefit from active code completion support as well.

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 explicit unit tests be generated instead. Although the palette interface conveyed the results of these tests immediately, participants remained adamant about explicit unit tests, likely due to concerns that future modifications to the generated code without reinvoking the palette might introduce bugs, or due to the desire to conform to standard testing practices. To support the generation of unit tests, however, the active code completion architecture must explicitly allow for code generation at locations other than at the cursor and handle the fact that statements generated after a code completion command are not necessarily encapsulated well enough to be immediately testable in isolation.

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 is 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 specifically 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 presentation of active code completion 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 familiar with editors like Vim that place a strong emphasis on keyboard shortcuts.

G. Minimal Impact on Editor 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 language and the Eclipse development environment. As we showed, a number of participants preferred other languages or editors. Many of these participants made comments asking that the features we describe not be restricted to one specific editor (e.g. NetBeans users) and some also called for support for other programming languages (e.g. Ruby and C#). 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

Our initial color palette was deemed overly complex by many participants (26), with many others (26) specifically saying that they would be satisfied with a simpler interface with colors listed next to their names. Other users wanted additional features, such as an "eyedropper" tool for selecting a color directly from an image on the screen. Similarly, the regular expression and SQL palettes were considered too simple by 12, and 15 participants respectively. User requests included syntax highlighting, a range of additional helpful tools for generating, testing and sharing regular expressions, and the ability to browse databases more directly. Due to the wide range of user needs even within a single class, it

may be that support for multiple palettes or a tabbed palette interface will be necessary in practice. To support incremental improvements to palettes based on 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 solicited suggestions from the participants of other classes that could benefit from an associated palette. 119 participants made one or more suggestion and we classified the suggestions into several categories. We also mention other suggestions made by researchers and developers via other channels in some cases, 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 [9] 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 [10]. 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 (1)

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 [11]. 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 highly interactive proof assistant interfaces.

V. Tool Design

Motivated by these use cases and with the design criteria developed as a result of our developer survey in mind, we sought to create an active code completion system. We named our project Graphite, an acronym for **Gra**phical **P**alettes to **H**elp **I**nstantiate **T**ypes in the **E**ditor. We describe the design decisions and trade-offs that went into this system in this section. A video demonstrating Graphite, including the process of writing a full palette, is available for further details [3]. The full source code of Graphite is also available at the URL given on the first page.

We chose to use Eclipse for the Java programming language because this combination was the most widely-used amongst participants in our survey. Eclipse plug-in development is well-supported and there are many examples available online that we can refer to, and the Eclipse project is available under an open source license.

A. HTML+Javascript based palettes

Eclipse is itself written in Java and uses the SWT and JFace graphical user interface toolkits. These frameworks are not widely used outside of the Eclipse ecosystem. As we have noted, 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. Indeed, many concepts (including regular expressions, colors and database queries) are similar across languages. To better support IDE-independence and language-independence, we chose instead to build palettes using HTML and Javascript. Javascript was among the most well-known languages in our study, just behind Java, C and C++, and is highly flexible. A number of user interface libraries are available, such as jQuery[4] (which we used to implement our palettes). All major browsers now feature sophisticated debuggers and run-time inspection facilities that ease debugging and testing. Deploying palettes using standard URLs is also simpler than attempting to integrate their source code directly into Java libraries and packages and eases the process of incremental development and deployment. All major windowing toolkits feature a web browser control that can allow for the display of these palettes within the editor environment. In Eclipse, we used the standard SWT Browser control.

B. Palette API

In order to allow palettes to communicate with the surrounding editor environment, a simple Javascript interface was injected. These functions were implemented by the editor plug-in, but appear as "native" browser functions to Javascript code. 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 appropriate amount of indentation is automatically inserted after any newlines in this string.
- `cancel()`
  Closes the palette without inserting a string.
- `getSelectedText()`
  Returns the text that is currently selected by the user in the editor, or an empty string if no text has been selected. This method is used to implement reinvocation.
- `getIDE()`
  Returns a string that specifies the IDE that is being used, "Eclipse" in our implementation.
- `getLanguage()`
  Returns a string that specifies the programming language that is being used, "java" in our implementation.

By limiting the complexity of this API, we reasoned that developers would be able to create specialized palettes more easily. An additional benefit is that the developers of other editing environments are also able to create plug-ins that support Graphite palettes with minimal effort.

To access this API, the palette must include a small script named `graphite.js` into their page. This file interacts with the host editor to create the graphite object and provides an alternative implementation of these methods when the palette is being tested in a web browser.

C. Palette Discovery

Graphite currently provides two methods for associating a palette with a particular type so that the Graphite plug-in is able to discover it when the code completion menu is invoked by a developer.

[3] http://www.youtube.com/watch?v=FfCoXQEHGAI

[4] http://jquery.com/

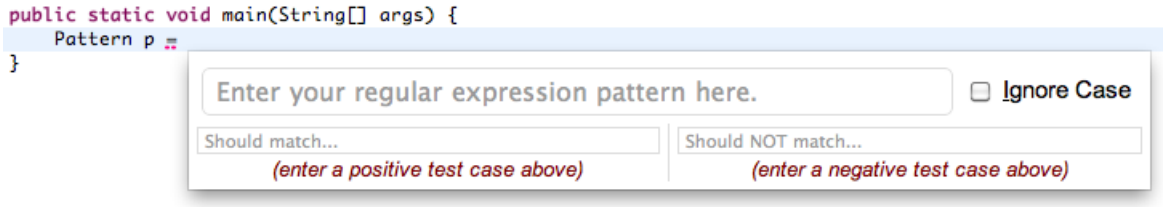

Figure 5. The regular expression palette developed using Graphite and used by subjects in the treatment group of the user study described in Sec. VI.

1) Annotation-based: For user-defined classes that the palette author 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 library providers to provide palettes that are specialized to their libraries and distribute them alongside their code. In this way, the users of the libraries are not required to discover that an external tool exists. If the user does not have the Graphite plug-in for Eclipse installed, the annotation has no effect.

2) Explicit: In cases where the palette developer cannot modify a class directly (such as palettes for classes in the Java standard library), a second method exists to explicitly associate a palette with the fully qualified name of a class. Currently, this feature is exposed as a preference pane in the Eclipse IDE. To facilitate automated installs in the future, a preferences file may also be added to the system.

D. Design Trade-Offs

The design that we have described is light-weight, highly flexible and highly responsive. It lays foundations for IDE and language portability. However, several trade-offs have also been made to achieve these benefits.

- Because palettes are implemented in Javascript, any differences between Java and Javascript become evident. For example, Javascript color names are, in two cases, slightly different from the corresponding Java color name. The Javascript regular expression engine is also slightly different. Although a Java applet could be embedded to provide necessary interoperability in cases where these differences are critical, this is more difficult than it would be if the palettes were implemented using the same language as the code.
- 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.
- Implementing a reinvocation mechanism that is robust to changes in the code and does not require the user to select the relevant information is difficult because this would require that the palette parse all the code to find relevant information. This also would require richer insertion APIs that allow for "dead" code to be removed. This puts a greater burden on palette developers and errors could affect user's code in subtle ways. The small inconvenience of having to select code before reinvoking a palette may be balanced by the knowledge that no other portions of the code will be modified inadvertently by that palette.

E. Palettes

We implemented two full-featured palettes to demonstrate and test the functionality of this framework: a regular expression palette (Figure 5) and a color selection palette (Figure 1).

1) 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 study of this palette.) A key describing this color scheme is displayed after the first test has been entered (not shown).

Users navigate between the three input areas 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. Because the palette is responsible for generating this literal, however, this is no longer a problem. In addition to the source code itself, the tests are retained in a comment beginning on the next line. The user can leave these as documentation or use them to create more formal unit tests within their preferred testing framework.

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. Although this is slightly inconvenient in that it requires that the user select the relevant portion of the code, it is the most robust method for implementing this feature – it can easily handle changes to the regular expression or tests and it does not require additional logic in the palette to determine which portions of the code are relevant, and should subsequently be replaced if a new regular expression is entered.

2) Color Selection: We omit a description of the full behavior of the Color selection palette due to space limitations.

VI. User Study Design

We conducted a comparative study to evaluate the usefulness and usability of a Graphite palette in a constrained setting – writing regular expressions. We used a between-subjects design by randomly splitting the participants into two groups: a control group, where subjects were not shown nor able to use the palette, and a treatment group, where subjects were shown a different palette and allowed to discover and use the regular expression palette if they wished.

A. Process

1) Preliminary Survey: The participants were first asked to fill out the pre-survey form about their prior experiences with various programming languages, integrated development environments (IDEs), and regular expressions, based on the previous survey that we had conducted. Of note, the subjects in this study were considerably less skilled with Java and regular expressions than in the previous survey, likely due to a sampling effect – our subjects were all PhD students at Carnegie Mellon.

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.

B. Within vs. between-subjects design

We decided to use 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 difficult. 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 hadn't used regular expressions recently.

C. Participants

We recruited 7 subjects from Computer Science Department and Institute for Software Research at Carnegie Mellon University. All the 7 subjects were PhD students. 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.

D. 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 online. We omit descriptions of each individual question due to space limitations.

1) Task Structure: The subjects were given a skeleton class which contained the instruction for how to complete the task and a method stub called `getRegexPattern` for each task. They are asked to fill the body of the `getRegexPattern` method. The `Task1.java` -

`Task9.java` files were listed in the Eclipse project explorer so that the subjects could easily open and reopen the tasks in any order that they wanted.

VII. User Study Results

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.

A. Control

In our previous online survey, we had found that no single strategy for writing regular expressions dominated the others. About 40% of participants indicated that they would use an external tool, 30% indicated a preference for writing a separate test script, and about 13% of participants chose the guess-and-check strategy. Although our sample size is too small to make a quantitative comparison, we did observe each of these strategies in the control group.

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 of any kind.

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.

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 (new), 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.

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 was able to fix the problem after that.

B. Treatment

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[5]), 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 factory pattern instantiation and also both had an initial error related to

[5]We did not observe any actual errors related to regex matching in our tool.

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 in class, 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 conclusive or 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.

VIII. Related Work

In addition to the references provided in the introduction, some other concepts 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* [12]. 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. [13]).

Editing environments like Barista [14] has previously been developed that merge concepts from both conventional text based and structured editors. Barista allows developers to use alternative code representations. It differs from Graphite in that these representations are built directly into the editor (although adding new representations is possible), and that Barista requires the users to use its own model-view-controller based 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 has a "show color dialog" [15] which allows the developers to launch a color picker directly from the code editor. Resharper [16] also has a color assistance tool which previews the actual colors right beside the code completion menu items. However, these IDE specific features are hard-coded – user-defined types cannot provide similar functionality. Recent versions of Visual Studio have started to explore user-defined palettes for widgets in the windowing system.

IX. 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 and generated a number of use cases and design constraints for this concept with an extensive survey of professional developers. Based on these findings, we developed an active code completion architecture that makes several novel design decisions to ease development, deployment and discovery of user-defined palettes. We created a palette for regular expression tasks and validated its usefulness with a user study. We claim that this provides evidence for the more

general claim that integrating palettes into a code editor is useful, and that active code completion systems like Graphite considerably ease this process.

ACKNOWLEDGMENT

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] D. Hou and D. Pletcher, "An evaluation of the strategies of sorting, filtering, and grouping api methods for code completion," in *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*, sept. 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 *2nd International Workshop on Domain-Specific Program Development (DSPD'08), co-located with OOPSLA'08*, Nashville, Tennessee, USA, 2008. [Online]. Available: http://hal.archives-ouvertes.fr/docs/00/35/02/66/PDF/6.pdf

[5] S. Han, D. R. Wallace, and R. C. Miller, "Code completion from abbreviated input," in *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 332–343. [Online]. Available: http://dx.doi.org/10.1109/ASE.2009.64

[6] 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.

[7] 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.

[8] J. Stylos, A. Faulring, Z. Yang, and B. A. Myers, "Improving API documentation using API usage information," in *VL/HCC*. IEEE, 2009, pp. 119–126. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/VLHCC.2009.5295283

[9] 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 *2010 Asia Pacific Software Engineering Conference (APSEC2010)*, Dec. 2010.

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

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

[12] T. L. Veldhuizen and D. Gannon, "Active libraries: Rethinking the roles of compilers and libraries," Oct. 05 1998, comment: 16 pages, 1998 SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing. [Online]. Available: http://arxiv.org/abs/math/9810022

[13] P. Miller, J. Pane, G. Meter, and S. Vorthmann, "Evolution of novice programming environments: The structure editors," 1994. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.110.3146;http://web.cs.cmu.edu/~pane/ftp/ILE.pdf

[14] Ko, A. J., Myers, and B. A., "Barista: An implementation framework for enabling new tools, interaction techniques and views in code editors," in *Proceedings of ACM CHI 2006 Conference on Human Factors in Computing Systems*, ser. Understanding programs and interfaces, vol. 1, 2006, pp. 387–396. [Online]. Available: http://doi.acm.org/10.1145/1124772.1124831

[15] "Coderush, show color dialog." [Online]. Available: http://documentation.devexpress.com/#CodeRush/

[16] "Resharper, color assistance." [Online]. Available: http://www.jetbrains.com/resharper/whatsnew/#Color