

SnipMatch: Using Source Code Context to Enhance Snippet Retrieval and Parameterization

Doug Wightman¹, Zi Ye¹, Joel Brandt², Roel Vertegaal¹

¹Human Media Lab, Queen's University
Kingston, ON, K7L 3N6, Canada
{wightman, zi, roel}@cs.queensu.ca

²Advanced Technology Labs, Adobe
San Francisco, CA 94103
joel.brandt@adobe.com

ABSTRACT

Programmers routinely use source code snippets to increase their productivity. However, locating and adapting code snippets to the current context still takes time: for example, variables must be renamed, and dependencies included. We believe that when programmers decide to invest time in creating a new code snippet from scratch, *they would also be willing to spend additional effort to make that code snippet configurable and easy to integrate*. To explore this insight, we built *SnipMatch*, a plug-in for the Eclipse IDE. SnipMatch introduces a simple markup that allows snippet authors to specify search patterns and integration instructions. SnipMatch leverages this information, in conjunction with current code context, to improve snippet search and parameterization. For example, when a search query includes local variables, SnipMatch suggests compatible snippets, and automatically adapts them by substituting in these variables. In the lab, we observed that participants integrated snippets faster when using SnipMatch than when using standard Eclipse. Findings from a public deployment to 93 programmers suggest that SnipMatch has become integrated into the work practices of real users.

ACM Classification: H.5.2. Information interfaces and presentation: User Interfaces—prototyping.

Keywords: Example-centric development, prototyping, natural language processing

INTRODUCTION

Programmers routinely search online for snippets to integrate into their source code [3, 5, 15, 22, 30]. This find-and-integrate behavior reduces the time to leverage Application Programming Interfaces (APIs), libraries, and algorithm implementations [5, 34]. Even when an API is well understood, snippets provide productivity gains: the time and effort required is often less than writing the code from scratch [3].

Despite the benefits of snippet use, substantial time can still be required to locate and integrate code snippets. Programmers often must locate and combine multiple code

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

UIST'12, October 7-10, 2012, Cambridge, MA, USA.
Copyright 2012 ACM 978-1-4503-1580-7/12/10...\$15.00.



Figure 1. The SnipMatch plug-in for the Eclipse development environment. A keyboard shortcut opens the search window (1) at the programmer's cursor position. Search results (2) are updated as the query is typed. The search query also affects integration: the local variable `playerScores` is included in the snippet (2). Snippet integration is previewed within the existing code.

snippets, rename or typecast variables, and manually locate and include dependencies [5, 7, 21]. After this complex integration has been performed, a code snippet may be discarded if it contains errors or otherwise does not work as expected [4]. Even a small decrease in the time required to perform this common task — finding and integrating snippets — could cause a qualitative change in behavior [12].

Existing code snippet solutions fall into one of two broad categories: those that leverage a carefully curated set of purpose-built snippets [9, 27], and those that either synthesize new snippets or mine relevant snippets from large repositories [3, 16, 23]. There is a simple trade-off associated with choosing one of these two approaches: curated collec-

tions yield higher-quality snippets, but are less complete. Mined or synthesized snippets can cover a much broader set of use cases, but may sometimes yield irrelevant, difficult to understand, or incorrect code.

Our research is driven by two insights:

First, given metadata about integrating a snippet into existing source code, we can build much more powerful tools for snippet retrieval and adaptation. In particular, we can use information provided in the search query, such as local variable names, to semi-automatically tailor snippets and facilitate further adaptation. We can also leverage local code context (e.g. variable types, imported libraries) to rank and filter results in relation to compatibility with the existing code. Unfortunately, gathering rich metadata about a snippet is difficult because users are usually focused on saving time when they are interacting with snippets. When a person is *creating* a snippet, however, she has already decided to invest time now in order to save time later.

Second, when a person creates a code snippet from scratch, she is willing to spend a reasonable amount of effort to make that code snippet configurable and easy to integrate. While creating a snippet, the author might add metadata using a snippet markup language. This is a viable method to obtain search and integration metadata: for example, variables to be renamed and preconditions to verify compatibility with the user’s existing code. To explore these insights, we built and evaluated *SnipMatch*, a search interface for finding and integrating curated code snippets.

This work offers three contributions:

1. A search algorithm for curated code snippets that leverages code context — Curated snippets are ranked, filtered, and customized based on the code in the development environment. SnipMatch builds on prior work in Integrated Development Environment (IDE) search [3, 8, 9, 18, 29]. The SnipMatch search algorithm extends the use of code context beyond the current programming language and framework to enhance the ranking of shared, curated code snippets. Our search algorithm uses the following features of the programmer’s source code to rank and filter prospective snippet results: variable types and names, the cursor position within the abstract syntax tree, program logic, and code dependencies. We selected these features because they can be used to determine how closely results match the existing code. Results that more closely match the features of the existing code – that is, make use of existing variables and require fewer modifications – are ranked higher.

2. A lightweight markup for specifying integration instructions for code snippets — Other tools suggest error-correction source code modifications [7, 8, 13]: for example, prompting the user to rename variables after a snippet has been inserted. These tools require human intervention because the intended use for a snippet can be ambiguous. Did the programmer intend to use the code snippet “as is”,

or only part of it? Should the snippet, or the existing code, be modified when there is an error? An integration tool requires more information to answer these questions.

3. Insights about how code snippet search tools are used, derived from the implementation and evaluation of SnipMatch. We implemented SnipMatch, a snippet search plug-in for the Eclipse IDE. To better understand how snippet search tools change the way people program, we conducted a comparative laboratory study with 16 participants and a public deployment with 93 programmers. We observed that SnipMatch was used to reduce context switching and as a memory aid. Participants reported that including snippet arguments in the search box was particularly effective for the two most common usage scenarios: shortcuts and quick reference.

RELATED WORK

Search Interfaces

Search interfaces [14, 26] can be tailored for specific tasks. Prior work includes systems to support data analysis [5, 24], web page revisitation [1, 31], and programming [3, 22]. Programming search interfaces can enhance web search results [15, 22] and, most relevant to our work, locate code snippets [3, 29]. Locating snippets can be time-consuming: in one study, 19% of programming time was spent looking for source code on the Internet [5].

Snippet search interfaces query code repositories to find *keyword* [3, 9, 11] and *structural* [2, 23, 29, 32] matches. Structural search can include building an internal representation of the program, while keyword matching is limited to text analysis. Google code search [11] locates occurrences of keywords in source code files. Uncommon, domain-specific search keywords can improve the relevance of results from this large repository. Blueprint [3] is a development environment plug-in that augments Internet keyword search queries with the language and framework used in the development environment. Snippets are automatically extracted from web pages and can be browsed within the plug-in. While keyword matching can be ineffective for locating many general code structure or program logic patterns, it is easy to use and can match comments, variables, dependencies, and other lexical features.

Structural snippet search interfaces perform static analysis to determine how code functions. For example, structural search interfaces can locate snippets that create an object of a certain type from other objects. Finding a Method Invocation Sequence (MIS) is a common API search task [29, 32, 16, 33]. PARSEWeb [32] ranks code fragments to be incorporated into MISs by frequency of use and length. S⁶ [28] allows programmers to include simple test cases and contracts with keyword searches. However, it can be difficult to ensure that snippets do not contain errors [23] and to differentiate between similar structural features: for example, frameworks and class libraries [2]. Further, Jungloid [23] and most other structural search interfaces are less effective when the programmer does not know the names of relevant classes, methods, or types.

SnipMatch is designed to provide more precise results and result rankings than these alternatives, but at a cost: someone must associate additional data with a code snippet for its search ranking and automatic integration to be improved. Snippet creators can also specify alternate search result wording to make snippets easier to find, mitigating the vocabulary problem [10]. SnipMatch allows programmers to enhance the accessibility of snippets that are important to them.

Snippet Integration

Many different types of modifications can be required to integrate a code snippet. EUKLAS [7] highlights source code errors and suggests corrections. EUKLAS can detect and correct missing JavaScript parameters, function and variable definitions, and imports. Eclipse Quick Fix [8] provides similar functionality for Java, and can also correct some unhandled exceptions. These tools allow programmers to quickly resolve many simple errors.

HelpMeOut [13] presents suggestions for correcting compiler and runtime errors. Relevant compiler error suggestions are found by examining the source code line referenced in the error. Relevant runtime exception suggestions are found by examining the stack trace. While HelpMeOut isn't specifically targeted at snippets, this approach could be useful for fixing snippet integration mistakes. Rather than fixing errors, SnipMatch attempts to prevent them from occurring.

The Codelets system [27] allows authors of example code to use a simple markup language to indicate which parts of an example are fixed and which parts the user should edit. Editable regions can be given names that are then referenced in custom user interfaces for configuring the example code. In SnipMatch, we extend this idea of providing users with a lightweight markup language for annotating snippets. In particular, we add syntax for expressing type information, integration instructions, and external dependencies. Additionally, we leverage this semantic information to make our code search more powerful.

Blueprint extracted over 100,000 code examples from blogs, forums, and human-created documentation available on the Internet [3]. In all of these cases, people expressly curated and published these snippets with the intention of sharing. SnipMatch is also dependent on programmers' willingness to curate and share snippets.

Sloppy Interpretation

Sloppy interpreters have lenient syntax requirements [25]. Inky [25] interprets keyword searches as web tasks and allows the user to click a button to perform them. For example, the top search result for the query "johnnd@gmail.com leaving the office now" might be "email johnnd@gmail.com about "leaving the office now". The keyword interpreter tokenizes the query, detects token types (e.g. email addresses), matches the tokens with pre-defined functions, and returns an ordered list of results. Koala [20] allows users to write scripts to automate web tasks by typ-

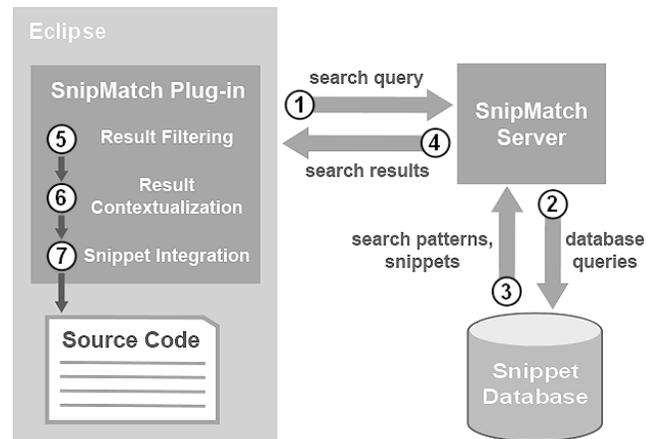


Figure 2. SnipMatch snippet search overview

ing sequences of commands. The interpreter can recognize the commands "type UIST into search field" and "click search button".

Koala built on previous work translating keyword commands into executable code [19]. This previous work was also the basis for a project that automatically generated code snippets from search queries [18]. Snippets are created by nesting methods, extracted from open source projects, with names similar to the search keywords. Unfortunately, this alternative to a code repository is often not practical because it can create semantically incorrect code.

SnipMatch does not generate snippets from extracted method calls. Similar to Inky, SnipMatch matches search queries with pre-defined snippets. This prevents semantic errors while preserving the primary benefits of sloppy interpretation: minimal wording and syntax requirements.

ENHANCING SNIPPET SEARCH WITH CODE CONTEXT

Programmers can use SnipMatch to find and integrate Java snippets from within the development environment. Pressing Ctrl-Enter while editing source code opens the SnipMatch search interface. The top search result is previewed inline as the programmer types in the search box. Figure 1 shows the result of typing "sort p" in the search box. The array `playerScores` is included in the top search result because this snippet has a *search pattern* with a parameter.

The SnipMatch search algorithm matches search queries to search patterns. A search pattern is a text description of the effect of the code snippet, interspersed with placeholders for snippet parameters. For example, this is the text description for the search pattern shown in Figure 1: "sort <array> in ascending order". "<array>" refers to a snippet parameter. To prevent type mismatches, the parameter type can also be specified in the search pattern. Snippets can have multiple search patterns.

When a snippet is submitted to SnipMatch, a lightweight markup language can be used to specify where search pattern parameters will appear in the snippet source code. This markup can also be used to import dependencies as required and define preconditions for snippets to appear in

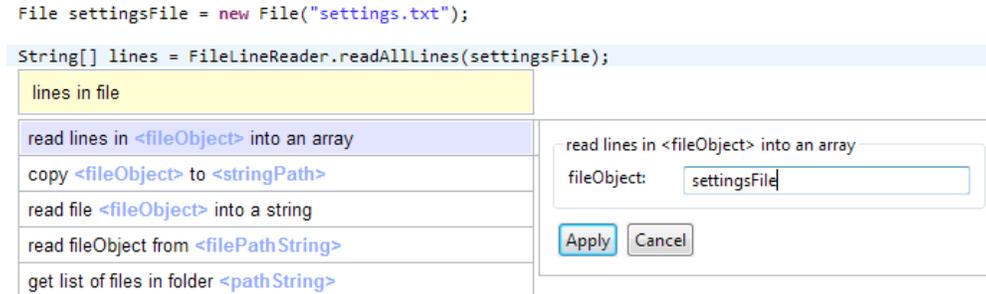


Figure 3. The argument editor (right window) is displayed if the user presses Enter before all arguments have been specified. This structured view prompts the user with text fields for each search pattern parameter. The user’s code is modified as the text field is completed.

the search results. After results are returned from the central server, the Eclipse plug-in filters and customizes the snippets based on the markup and the code context in the development environment (Figure 2).

SnipMatch snippets can be private or public. Public snippets are available to all users, while private snippets appear only in the snippet creator’s search results. To ensure correctness and maintain a consistent level of snippet quality, system moderators review snippets that are added to the public repository.

Design Rationale

We are interested in reducing the time and effort required to both *locate* and *integrate* code snippets. As described in the introduction, snippet integration can be complicated and time-consuming. Previous approaches for supporting snippet integration include methods that prompt the user with options [7, 8, 13] and methods that generate snippets from keywords based on rules [18, 19]. Although helpful to highlight problems that require attention, user prompts still require code comprehension, decision-making, and manual input prior to testing. Generative methods frequently create semantic errors.

One viable alternative that does not require manual input at the time of integration is for programmers to annotate snippets *when they are created* with additional machine-readable markup language. This information can then be used by the search interface to provide enhanced search ranking and snippet integration. Programmers already create Eclipse Templates for personal use, and share and comment on code snippets online [11, 22]. We believe that programmers will similarly contribute search patterns and integration markups if they will improve snippet usability. As with Wikipedia, relatively few individuals need to contribute for the system to be useful for many.

IMPLEMENTATION

The SnipMatch plug-in was written in Java using the Eclipse Plug-in Development Environment (PDE). The plug-in communicates with the server through HTTP requests. Server-side, the search algorithm is written in C++ and PHP, and snippets are stored in a MySQL database.

During the evaluation, our server was located in the United States.

Snippet Search

Search Window. The search window (Figure 1) has two main components: a static text field for the search query input, and a dynamically resizing region below for search results. The ordered list of search results is updated as the contents of the text field are modified. This provides the user with instant feedback, facilitating snippet discovery and step-wise refinement of search queries. To increase readability, search results have basic syntax highlighting; different font colors are used for keywords, arguments, and placeholders for missing arguments.

Argument Editing. If a user attempts to insert a snippet with missing arguments, the argument editor dialog appears to the right of the search window (Figure 3). The dialog presents a structured view of the snippet parameters and includes text fields for entering the arguments. Changes to the arguments are immediately reflected in the source code. The editor can also be manually invoked for any highlighted result by pressing Ctrl-Enter.

Tab Completion. Since search results are updated as the search query is typed, the user can refine the search query based on the results. For example, the search query “read” might return some results that begin with “read lines from file”, and other results that begin with “read character from keyboard”. The user can narrow the results to only include those pertaining to file operations by changing the search query. Tab completion facilitates this process. When the user presses Tab, the search query text is autocompleted up to the next parameter in the currently highlighted result. For example, if the search query is “read”, and the highlighted result is “read lines from file <filePath>”, pressing Tab will change the search query to “read lines from file ”.

Snippet Icons. A small group of icons is docked in the lower right corner of the currently highlighted result (Figure 5). Most of these icons are buttons that allow the user to send feedback about the result. Feedback options include rating, flagging, and commenting. All user feedback is logged for manual analysis. A yellow warning icon appears when the snippet includes changes to the source code that are omitted

from the preview, such as helper classes. Resting the mouse over this icon reveals a summary of all the hidden changes.

Snippet Submission

The SnipMatch Eclipse plug-in includes an interface for adding and editing code snippets. This interface allows users to modify search patterns, and includes features to facilitate the addition of integration markups to the snippet code. SnipMatch uses the Java type hierarchy to recognize standard and user-created parameter types that appear in search patterns.

Search Results

There are three different types of search results: *in-order*, *unordered*, *unsigned*. After search results are found, they are ranked, filtered, and shown to the user in a single list.

In-order. An in-order result is a snippet with a search pattern that begins with exactly the same sequence of characters as the search query. For example, the result “read from file <filePath>” is an in-order result for the search query “read from”.

Unordered. An unordered result is a snippet whose search patterns do not begin with the search query, but do contain one or more tokens from the search query. For example, the result “read from file <filePath>” is an unordered result for the search query “file”.

Unsigned. An unsigned result is a snippet that does not yet have a search pattern, but whose code contains one or more tokens from the search query. These snippets can be retrieved not only from the SnipMatch snippet database, but also from external snippet repositories.

Result Ranking

Results are first ranked by type. In-order results appear first, followed by unordered. Unsigned results appear after all other results. In addition to being ranked by type, each result type uses a different set of ranking criteria.

In-order. In-order results are first ranked by the number of search query tokens matching each search pattern. Next, results are ranked by the number of missing arguments. Results with fewer missing arguments are ranked higher.

Unordered. Unordered results are ranked based on term frequency-inverse document frequency (tf-idf) [17]. Tokens from the search query are matched against search patterns. Search patterns that contain more query tokens or query tokens that appear less frequently in other search patterns are ranked higher. User feedback (e.g. votes and frequency of selection) will be incorporated in the future.

Unsigned. The same ranking criteria as for unordered results are applied, except that the search query is matched against the snippet code instead of the search patterns.

Result Filtering and Contextualization

Since the SnipMatch server has limited information about the user’s code, the client modifies the returned search results before they are shown to the user. This is completed in two stages: result filtering and result contextualization.

```
$(import(java.io.File))
$(import(java.io.BufferedReader))
$(import(java.io.FileReader))
$(import(java.io.IOException))
$(import(java.util.ArrayList))

$(helper)
class FileLineReader {
    public static String[] readAllLines(File file) {
        ArrayList<String> lines = new ArrayList<String>();
        try {
            BufferedReader br = new BufferedReader(new FileReader(file));
            String line = null;

            while ((line = br.readLine()) != null) {
                lines.add(line);
            }
            br.close();
        }
        catch (IOException e) { return null; }
        return lines.toArray(new String[lines.size()]);
    }
}
$(endHelper)

String[] ${freeName(lines)} = FileLineReader.readAllLines(${fileObject});
```

Figure 4. Integration markup

Filtering. Since the SnipMatch server does not have access to the Java type hierarchy, some search results may contain incompatible types. The client performs a type compatibility test on each search result and filters out the results that fail the test. For example, the query “for x” may return a result that tries to create a for loop using a hypothetical local integer x. However, if the client detects a local variable x of another type, this search result will be removed. The client also filters results that are incompatible with the existing code. For example, if integration markup preconditions are not met.

Contextualization. If the user omits an argument, or does not finish typing an argument, the client analyzes the user’s source code to find variables within the current scope that can serve as arguments. This list of possible substitutions is used to create variations on the original, incomplete search result. These client-generated results are presented to the user instead of the original. In Figure 1, the client detects a compatible local variable (playerScores) and uses it to complete the server’s search results. After variables are matched with parameters, the snippet is customized according to the integration markup.

Snippet Integration

Instructions for integrating a snippet into the user’s source code are expressed using a lightweight markup embedded within the snippet code. The SnipMatch markup is similar to the markup used in Eclipse Templates [9], with several additions. Eclipse Templates is a built-in feature in Eclipse that allows users to create and insert code “templates” (snippets). Like SnipMatch, it allows snippets to be integrated into the source code by taking into account local variables and adding missing import statements. Unlike SnipMatch snippets, each Eclipse Template can only be found through its single-word name. Parameters, helper

classes, and precondition checking are not supported. Figure 4 shows the snippet code associated with the search pattern “read lines in <fileObject> into an array”. This snippet includes two additions to the Eclipse Templates markup: a search result argument (<fileObject>) and a helper class. In figure 4, `#{fileObject}` indicates where to insert the argument for the <fileObject> parameter. The helper class is delimited by `#{helper}` and `#{endHelper}`. Helper classes are inserted in the current source file.

Helper classes are an optional feature that can make code easier to read and reduce redundancy. For example, a programmer might prefer to read text from a file by calling a method of a helper class instead of having the method code appear inline. SnipMatch maintains a record of inserted helper classes so that future snippet insertions will not create duplicate classes.

Another addition to the Eclipse Templates markup is support for integration preconditions. The SnipMatch markup includes the terms `#{startPrecondition}` and `#{endPrecondition}`. The code between these two terms is expected to be a Java method named `preconditionTest`. Before the snippet is shown to the user, the `preconditionTest` method is executed in a secure sandbox that prevents access to other local processes or communication with other machines. The return value of the method is a Boolean that determines whether or not the user’s code meets the necessary conditions for the snippet to be integrated. This method accepts three arguments with information about the source code file currently open in the IDE. The first argument is an instance of the Eclipse JDT ASTParser class. This argument contains a copy of the abstract syntax tree that can be traversed to perform validations. A copy of the source code and the current cursor position in the file are also included as arguments – for example, so that a snippet creator can alternately write a simple grep when string matching is sufficient.

A graphical interface for specifying preconditions without writing code could be created. For now, video tutorials on the SnipMatch website and an interface for selecting markup within the snippet editor provide usage instructions and examples.

Benefits of Client-Server Search Processing

Allowing both the client and the server to process search results has privacy and efficiency benefits. It mitigates privacy concerns, since source code is not sent from the client to the server. Computation is distributed between the client and server, with the client handling the computationally expensive filtering and integration steps that are dependent on the user’s code. Server results can be cached because they are not specific to the user’s code.

Extensibility

Extending SnipMatch to support other imperative programming languages is straightforward. In addition to Java, we have tested SnipMatch with C++, PHP, and JavaScript. The search algorithm and markup were not changed. The

Eclipse plug-in was modified to extract variable names and types from these different ASTs. Type consistency checking was disabled for the dynamically typed languages, but search pattern parameters were still supported. Wrapper classes have been created to simplify the process of extending the plug-in for third-party developers.

EVALUATION

We conducted three studies: a lab study, an analysis of usage logs following the public deployment of SnipMatch, and interviews with programmers who have used SnipMatch. The lab study was conducted to gather initial feedback and assess ease of use. We then made SnipMatch publicly available and analyzed our logs to gain additional insights and confirm external validity. Finally, we interviewed five SnipMatch users to learn more about their needs and search behaviors.

STUDY 1: EVALUATING SNIPMATCH IN THE LAB

Method

Two sets of 8 computer science students were recruited for this study. Each participant was given two programming tasks to complete. Participants in the first set were given a brief (5-minute) SnipMatch tutorial and asked to use SnipMatch instead of searching online for code snippets. Participants in the second set were allowed to search online and were not trained to use SnipMatch.

To avoid priming our participants with search keywords, we explained the tasks by showing images. For the first task, we showed two images. The first image depicted two file folders, one full of files and the other empty. It was described as the “before” image. The second image contained the same two file folders with the all of the files now in the other folder. Participants were asked to “write a program to change the current state of the system to the second image”. They were also shown that the folder that contained the files in the first images existed on the computer and was currently full of files. For the second task, we showed one image, depicting a simple Graphical User Interface (GUI) for moving files from one folder to another. This GUI included two text fields, labeled “Source folder” and “Target folder”, and a button labeled “Move files”.

At the time of the study, SnipMatch included 29 snippets and 55 search patterns. The snippets included all of the functionality required to complete the tasks. The larger number of search patterns indicates that some snippets were discoverable through more than one search pattern. Approximately half of the snippets were created specifically for this study. Drawing from textbooks and our experiences teaching computer science, we attempted to include snippets to support most standard file input and output operations and many basic GUI features. To narrow the scope of our investigation to features specific to SnipMatch, we did not include search results from external code repositories, such as Google Code Search.

We anticipated that participants using SnipMatch would need to perform a minimum of 7 snippet integrations: 2 for

the first task and 5 for the second. This calculation was based on the assumption that participants had memorized the method calls required to complete our tasks. It also assumed that participants do not use SnipMatch for basic programming statements (e.g. to create a for loop) and that they prefer to copy and paste their code rather than use SnipMatch repeatedly.

After completing the tasks, each participant using SnipMatch filled out a questionnaire. We logged all SnipMatch searches and the times required by the server and the client to generate the results.

We recorded web search queries and web pages visited for the participants who did not use SnipMatch. After both tasks were completed, we asked these participants about their programming behavior and experiences working with code snippets found online.

Results

Programming with SnipMatch. All of our participants successfully completed both tasks. On average, participants completed the first task in 12 minutes (*s.e.* 1.5) and the second in 28 minutes (*s.e.* 3.3). On average, participants opened the SnipMatch search window 14.2 times (*s.e.* 2.2) and selected a snippet to integrate 74% of the time. Participants performed many exploratory searches to test the capabilities of the system. Most also used SnipMatch to write the loop required in the first task and to create the multiple labels and textboxes for the second task. Verbs were popular search terms (copy, create, move, open, read).

Two participants mentioned that they did not need to remember syntax. One of these participants explained the process of writing code using SnipMatch as “search to figure out how to do something, resulting in the creation of an object, then search again, including this object, to continue using it”. Several participants indicated that they were interested in using SnipMatch to create private snippet repositories.

The mean rating for the question “I would use this tool on a regular basis if it was available in my preferred development environment” is particularly encouraging ($\mu=4.75$ on 5 point Likert Scale). Participants also consistently gave SnipMatch high marks as an efficient alternative to online search ($\mu=4.75$) and for ease of use ($\mu=4.5$).

Programming without SnipMatch. All participants successfully completed the first (file i/o) task. Two participants were unable to complete the second (GUI) task within an hour. Including only data from tasks that were completed, on average participants performed 14 online searches (10 min., 18 max.), 7 for each task. On average, participants viewed 9 (5 min., 12 max.) non-search engine web pages for the first task and 15 (8 min., 21 max.) for the second task. Participants completed the first task in 25 minutes (*s.e.* 2.5) and the second task in 37 minutes (*s.e.* 3.6). All searches were performed on Google, with the exception of four performed on Stack Overflow.

Discussion

Programming with SnipMatch. Study participants understood how to use the tool and were able to use it effectively. We were surprised that all participants successfully completed both tasks. One participant had never programmed in Java and GUI programming can be difficult. Most participants chose to use SnipMatch even when it wasn't necessary. For example, they used it to create the loops and as an alternative to copying code from elsewhere in the file.

One participant began the first task by creating String variables for the directory paths. With the variables in scope, he then typed his first search query and the top search result previewed the exact code required to complete the first step for this task: creating an array of File objects for the files in the directory identified in one of his Strings. Several participants regularly took advantage of this context-sensitivity, adopting a search-based code writing behavior in which objects created from prior searches were included as keywords in future searches.

Some results were not as accurate. Participants requested that synonyms be added to the system. In particular, it was suggested that words referring to the same abstract concept in different programming languages might be interchangeable (e.g. form and frame) for search purposes. Also, preconceived ideas regarding command lines initially biased some participants towards selecting snippets that included string arguments instead of object arguments. This was overcome once SnipMatch displayed results that included local object variable names.

Participant feedback indicates that they were comfortable including arguments within search queries. The inline snippet preview provides code context for the arguments. We observed several participants reading the preview as they entered arguments.

The combination of natural language search results and inline previews was sufficient for participants to understand most snippets before integration. Although snippet integration can be undone with a single undo operation, this option was rarely used.

The average server response time was 64ms (*s.e.* 2.0). The Eclipse plug-in then spent 172ms (*s.e.* 15.9), on average, customizing the server results. While the code is not optimized, we believe that these numbers reflect the substantial offloading of computationally expensive operations to the client. Operations specific to the programmer's existing source code are performed client-side. This preserves privacy. With this approach, it is also possible to cache all server responses. SnipMatch can be provided to a large number of users at relatively low cost, similar to hosting static HTML pages.

Programming without SnipMatch. Only one participant who did not have access to SnipMatch completed the tasks in less time than the slowest participant who used SnipMatch (14 minutes, 19 minutes). These numbers are encouraging,

but this comparison has many limitations, including the small number of snippets that were in the SnipMatch database. While we expect that SnipMatch will continue to perform well as the number of snippets increases – since search patterns are short, precise, and can be refined as required – in this study, we are more interested in the differences in search behaviors.

On average, participants performed exactly the same number of search queries (14). However, participants using SnipMatch spent less time finding and integrating snippets. Without SnipMatch, participants were not able to directly select snippets from the search result list. Participants opened and viewed many additional web pages (24 on average) listed in the search results. These web pages contained Java examples, tutorials, and class documentation.

Most participants opened multiple tabs in the web browser and flipped between tabs, comparing examples. When asked about this behavior, participants explained that they were checking for differences in dependencies, attempting to verify that the code would operate as expected, and determine which of the examples would be easiest to integrate. With SnipMatch, participants often did not dwell on the search results. They typically inserted snippets with little hesitation, then experimented inline. We believe that this was partially due to the smaller time commitment required to insert snippets with SnipMatch. Viewing the snippet inline, participants could also benefit from the error and warning highlighting provided by the IDE.

Without SnipMatch, participants frequently revisited search result pages and selected alternate links rather than performing additional searches. After completing the tasks, 4 of our 8 participants indicated that they had difficulty coming up with alternate wording that would “make a difference” to the search results shown. They explained that adding additional words to their search queries often did not improve the search results. SnipMatch users may be less likely to experience this problem, since search queries are matched against search patterns, rather than whole documents, for *in-order* and *unordered* results.

Limitations to the Participant Recruitment Method

While we were careful to recruit participants from the same population (computer science students at our university) and using the same recruitment channel (our faculty mailing list), the first set of participants was recruited before the second set of participants. We initially intended only to obtain first use data, then decided to expand the study. Between sets, participants had similar programming experience (first set: 3-10 years, second set: 2-10 years), age (20-29, 21-26), and gender balance (1 female, 2 female). No statistical tests were performed on the study results.

STUDY 2: DEPLOYMENT TO 93 PROGRAMMERS

Method

We made the SnipMatch plug-in publicly available to gain additional insights and verify external validity. We were specifically interested in better understanding usage in the

wild. Will programmers use SnipMatch? How often will programmers use SnipMatch? We were also interested to determine the types of search queries and snippets that are most frequently used.

To obtain users, we created a webpage and embedded links to share it on Facebook, Twitter, and Google+. We then publicly announced SnipMatch on academic and industry mailing lists. With user consent, we logged all search queries and snippet insertions conducted during the first three weeks of public deployment. Before the announcement, we increased the number of SnipMatch snippets (72 snippets, 111 search patterns).

Results

93 programmers performed 516 searches: 345 resulted in snippets being inserted into the existing source code, and 171 were cancelled. Ten programmers used SnipMatch on more than four days. Four programmers used SnipMatch during each of the three weeks. These programmers performed 22-54 search queries. Five programmers created snippets. Two of these programmers are among those who have used SnipMatch on the largest number of days. Programmers created snippets for logging, class creation, object and value comparison, and debugging tasks. To preserve external validity, these results exclude usage by all individuals associated with the research and development of SnipMatch.

Discussion

Programmers are often eager to try new tools, but long-term retention and integration into daily practice is substantially less common. As a comparison point, Blueprint had a 1% user retention rate over a five-month span (where “retention” was measured by *any* use of the tool five or more months after initial installation) [3]. In the unlikely case that SnipMatch can maintain even a moderate portion of its current 4% (4/93) rate of *weekly* use, we will consider the tool to be highly successful.

When we conducted phone interviews (described in the following section) we were surprised to learn that at least two of our regular users are professional programmers, and that they are using SnipMatch in the workplace. Programmers with no direct interest in our work are regularly using SnipMatch while creating commercial software. In particular, this professional interest demonstrates need for a snippet search and insertion tool.

The snippets that print values and perform type conversions were among the most frequently inserted, along with several of the user-created snippets. These popular print and conversion snippets have search patterns with parameters. Some of the frequently used user-created snippets also have parameters and other features that require the snippet markup. This indicates that programmers were able to teach themselves to perform searches and use the markup.

STUDY 3: INTERVIEWS WITH PROFESSIONALS

Method

We sent an email to each of the 49 SnipMatch users who registered for an account, requesting that they participate in a 15-minute phone interview. 5 of our users, all professional programmers, agreed to be interviewed (all male, two living in the United States, one in Germany, and two in India). Two of these individuals are among the four programmers who have used SnipMatch during each of the three weeks of public deployment. We asked our interviewees to describe situations when they had used SnipMatch. We also examined their usage logs and spoke with them about searches they had performed. Then we asked if they had any difficulties using SnipMatch or ideas for improving the tool.

Results and Discussion

Common Usage Scenarios. Our interviewees described two common usage scenarios: shortcuts and quick reference. All interviewees used SnipMatch as a typing shortcut. For example, typing “convert”, “log”, or “println”, along with arguments, in the search box. In addition to saving time by reducing keystrokes, one programmer reported that this helped him to “focus his attention”, “staying in flow with the code”. Three programmers described specific situations when they had inserted snippets that they could not have written from memory, including design patterns and API calls. One programmer explained that SnipMatch was particularly useful for his Android development, which he found to require many “hard to remember structures”. Another mentioned writing snippets so that he would not need to memorize how to call JDBC methods. These programmers did not memorize how to type the search queries to insert these snippets – they trusted that they could find them quickly.

SnipMatch complements libraries. Each call requires some boilerplate code: minimally, importing a library and passing arguments. SnipMatch makes this easier, and shifts the balance towards reuse over code duplication. Further, with SnipMatch, programmers are able to help each other maintain good coding practices by curating the search results: if a snippet duplicates code instead of importing it, users can vote, flag, or comment to provide feedback.

Suggestions for Improvement. The programmers in India both requested that we reduce the server response time. In addition to promising to add a geographically proximate server, we proposed to modify the client to cache most frequently used snippets. This will also allow SnipMatch to function offline.

One programmer requested that search results for recently added snippets appear when the search box is empty. He was concerned that users might otherwise not discover them. For example, if a user had previously searched for image manipulation snippets and hadn’t found any, he thought they might not think to search again for some time. We proposed to implement his suggestion and also to up-

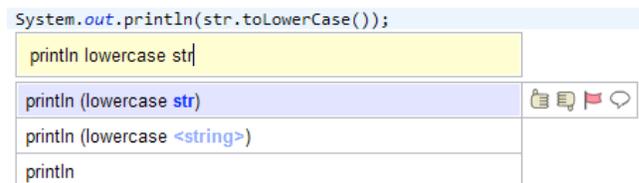


Figure 5. A search result with a nested snippet.

date the client to display results from other snippet databases (i.e. *unsigned* results) below SnipMatch results.

A programmer who uses SnipMatch regularly requested a method for specifying Boolean expressions that could be varied from within the search box. He indicated that he didn’t want to have to create snippets for each combination. We then told him about the nesting feature that we are building, not yet released for public use. He thought it would be sufficient for his purposes.

INSERTING MULTIPLE SNIPPETS WITH ONE QUERY

During the lab study, participants spent substantial time manually integrating snippets. For example, most participants in both studies inserted one snippet to retrieve a list of the files in a directory and a second to move files between directories. The participants then modified the snippets to interact. To reduce the time and effort required to integrate snippets, we designed an extension to SnipMatch that allows programmers to insert multiple snippets with a single search query (Figure 5).

For example, if `file1` is a Java File object, SnipMatch can recognize that the search query “move file1 to file1 parent directory” can be satisfied by combining snippet results that match the following search patterns: “move <x> to <y>” and “<z> parent directory”. In this example, <x>, <y>, and <z> represent search pattern parameters. To reduce ambiguity in the search results, parentheses are inserted: “move file1 to (file1 parent directory)”.

To implement this SnipMatch extension, we wrote a context-free parser that interprets search pattern parameters as nonterminal symbols and generates production rules from search patterns. We also added a new, optional, field to store the snippet return type. The left hand side of each production rule is a nonterminal symbol representing the return type of the snippet. This allows programmers to specify, by writing in-order search queries, how multiple snippets are to be combined.

Nested search results – search results that are produced by this extension – are first ranked using the *in-order* result ranking criteria. Next, the results are ranked by nesting depth. Results that have fewer levels of nesting are ranked higher. The ranked list of *nested* results appears below the *in-order* results and above *unordered* results in the search result list.

When performing a nested search, programmers will likely not type the entire query at once. Instead, they might add additional words to a shorter query as results appear. For example, in an informal evaluation of this extension we

observed a programmer building up to the search result “read file (lowercase <string>) into a string”. He first typed “read file”, then remembered that his code could be storing the file name in the wrong case, and decided to modify the search query to “read file lower” to verify that SnipMatch supports this nesting. While this extension has not been formally evaluated, we believe that it is a useful starting point for future investigations.

CONCLUSION

We have documented initial user experiences that demonstrate how search patterns and the snippet markup can improve a snippet search interface. Results from our lab study and public deployment suggest that SnipMatch can be an effective tool for certain tasks. Professional programmers, found to be using SnipMatch in the wild, reported that SnipMatch was useful as a memory aid and reduced context switching.

Unlike prior search tools for curated snippets, results are ranked and filtered based on both a snippet markup and existing code in the IDE. SnipMatch users can include local variables in search queries to pass them as arguments, customizing curated snippets from the search box.

Installation instructions and a quick start tutorial are available at <http://snipmatch.org>. We have released SnipMatch as open source. The source can be downloaded from the Eclipse Foundation’s code repository. Future work includes two main activities: a longitudinal field study and extending the markup to support refactoring.

REFERENCES

1. Adar, E., Dontcheva, M., Fogarty, J., and Weld, D. S. Zoetrope: Interacting with the Ephemeral Web. In *Proc. UIST*. pp. 239-48, 2008.
2. Bajracharya, S., T. Ngo, et al. Sourcerer: A Search Engine for Open Source Code Supporting Structure-Based Search. In *Companion to OOPSLA: ACM SIGPLAN*. pp. 681-82, 2006.
3. Brandt, J., Dontcheva, M., Weskamp, M., Klemmer, S.R. Example-Centric Programming: Integrating Web Search into the Development Environment. In *Proc. CHI*. pp. 513-522, 2010.
4. Brandt, J., Guo, P. J., Lewenstein, J., Dontcheva, M., Klemmer, S. R. 2009. Opportunistic Programming: Writing Code to Prototype, Ideate, and Discover. *IEEE Software*, 26 (5), 18-24.
5. Brandt, J., Guo, P.J., Lewenstein, J., Dontcheva, M., Klemmer, S.R. Two Studies of Opportunistic Programming: Interleaving Web Foraging, Learning, and Writing Code. In *Proc. CHI*. pp. 1589-1598, 2009.
6. Dontcheva, M., Drucker, S. M., Salesin, D., and Cohen, M. F. Relations, Cards, and Search Templates: User-Guided Web Data Integration and Layout. In *Proc. UIST*. pp. 61-70, 2007.
7. Dörner, C., Myers, B. A. EUKLAS, Plug-in for Eclipse IDE. <http://www.cs.cmu.edu/~euklas>
8. Eclipse Quick Fix. http://wiki.eclipse.org/FAQ_What_is_a_Quick_Fix%3F
9. Eclipse Templates. <http://www.ibm.com/developerworks/opensource/tutorials/os-eclipse-code-templates/index.html>
10. Furnas, G. W., Landauer, T. K., Gomez, L. M., and Dumais, S. T. The Vocabulary Problem in Human-System Communication. In *Communications of the ACM* 30, 11 (Nov 1987), 964-971.
11. Google Code Search. <http://www.google.com/codesearch>
12. Gray, W. D. and D. A. Boehm-Davis. Milliseconds Matter: An Introduction to Microstrategies and to Their Use in Describing and Predicting Interactive Behavior. *Journal of Experimental Psychology: Applied* 6(4). pp. 322-35, 2000.
13. Hartmann, B., MacDougall, D., Brandt, J., and Klemmer, S. What Would Other Programmers Do? Suggesting Solutions to Error Messages. In *Proc. CHI*. pp. 1019-1028, 2010.
14. Hearst, M. *Search User Interfaces*, Cambridge University Press, Cambridge, UK, 2009.
15. Hoffmann, R., Fogarty, J., and Weld, D. S. Assieme: Finding and Leveraging Implicit References in a Web Search Interface for Programmers. In *Proc. UIST*. pp. 13-22, 2007.
16. Holmes, R., Murphy, G. C. Using Structural Context to Recommend Source Code Examples. In *Proc. ICSE*. pp. 117-125, 2005.
17. Jones, K. S. A Statistical Interpretation of Term Specificity and Its Application in Retrieval. *Journal of Documentation* 28: 11, 1972.
18. Little, G. and Miller, R.C. Keyword Programming in Java. In *Proc. ASE*. pp. 84-93, 2007.
19. Little, G. and Miller, R.C. Translating Keyword Commands into Executable Code. In *Proc. UIST*. pp. 135-144, 2006.
20. Little, G., Lau, T.A., Cypher, A., Lin, J., Haber, E.M., Kandogan, E. Koala: Capture, Share, Automate, Personalize Business Processes on the Web. In *Proc. CHI*. pp. 943-946, 2007.
21. M. Kim, L. Bergman, T. Lau, and D. Notkin. An Ethnographic Study of Copy and Paste Programming Practices in OOPL. In *Proc. ESEM*. pp. 83-92, 2004.
22. Mamykina, L., Manoim, B., Mittal, M., Hripesak, G., Hartmann, B. 2011. Design Lessons from the Fastest Q&A Site in the West. In *Proc. CHI*. pp. 2857-2866, 2011.
23. Mandelin, D., Xu, L., Bodik, R., and Kimelman, D. Jungloid Mining: Helping to Navigate the API Jungle. In *Proc. PLDI*. pp. 48-61, 2005.
24. Medynskiy, Y., Dontcheva, M., and Drucker, S. M. Exploring Websites through Contextual Facets. In *Proc. CHI*. pp. 2013-22, 2009.
25. Miller, R. C., Chou, V. H., Bernstein, M., Little, G., Van Kleek, M., Karger, D., and Schraefel, M. Inky: A Sloppy Command Line for the Web with Rich Visual Feedback. In *Proc. UIST*. pp. 131-140, 2008.
26. Morville, P., and Callender, J. *Search Patterns: Design for Discovery*. O’Reilly Media, Inc., 2010.
27. Oney, S., Brandt, J. Codelets: Linking Interactive Documentation and Example Code in the Editor. In *Proc. CHI*. 2012.
28. Reiss, S. P. Semantics-Based Code Search. *ICSE* 2009.
29. Sahavechaphan, N. and Claypool, K. XSnippet: Mining for Sample Code. In *Proc. of OOPSLA*. pp. 413-30, 2006.
30. Stylos, J. and Myers, B. A. Mica: A Web-Search Tool for Finding API Components and Examples. In *Proc. VL/HCC*. pp. 195-202, 2006.
31. Teevan, J., Cutrell, E., et al. Visual Snippets: Summarizing Web Pages for Search and Revisitation. In *Proc. CHI*. pp. 2023-32, 2009.
32. Thummalapenta, S. and Xie, T. PARSEweb: A Programmer Assistant for Reusing Open Source Code on the Web. In *Proc. ASE*. pp. 204-13, 2007.
33. Xie, T., Pei, J. MAPO: Mining API Usages from Open Source Repositories. In *Proc. MSR*. pp. 54-57, 2006.
34. Yeh, R. B., Paepcke, A., and Klemmer, S.R. Iterative Design and Evaluation of an Event Architecture for Pen-and-Paper Interfaces. In *Proc. UIST*. pp. 111-120, 2008.