Example-Centric Programming: Integrating Web Search into the Development Environment

Joel Brandt[1,2], Mira Dontcheva[2], Marcos Weskamp[2], Scott R. Klemmer[1]

Stanford University HCI Group
Computer Science Department
Stanford, CA 94305
{jbrandt, srk}@cs.stanford.edu

Advanced Technology Labs
Adobe Systems
San Francisco, CA 94103
{mirad, mweskamp}@adobe.com

ABSTRACT

The ready availability of online source code examples has fundamentally changed programming practices. However, current search tools are not designed to assist with programming tasks and are wholly separate from editing tools. This paper proposes that embedding a task-specific search engine in the development environment can significantly reduce the cost of finding information and thus enable programmers to write better code more easily. This paper describes the design, implementation, and evaluation of Blueprint, a Web search interface integrated into the Adobe Flex Builder development environment that helps users locate example code. Blueprint *automatically augments queries with code context*, presents an *example-centric view of search results*, *embeds the search experience into the editor,* and retains a *link between copied code and its source*. A comparative laboratory study found that Blueprint enables participants to write significantly better code and find example code significantly faster than with a standard Web browser. Log analysis from a three-month field deployment with 2,024 users suggested that task-specific search interfaces may cause a fundamental shift in how and when individuals search the Web.

ACM Classification: H5.2 [Information interfaces and presentation]: User Interfaces—*prototyping*.

General terms: Design, Human Factors

Keywords: Example-centric development

INTRODUCTION

Programmers routinely face the “build or borrow” question [7]: implement a piece of functionality from scratch, or locate and adapt relevant existing code? The increased prevalence of online source code—shared formally in code repositories and documentation [1] and informally in blogs and forums [9]—is fundamentally reducing the cost of borrowing code. [6, 22]. The Web now contains example code that implements nearly any piece of routine functionality [5], making it possible for programmers to opportunistically build applications by iteratively searching for, modifying, and combining examples [8, 15].

Borrowing code is part of a diverse set of Web tasks completed by programmers. Brandt and colleagues found that programmers’ use of the Web varies greatly with their intended goal [6]. For example, when users are interested in *just-in-time learning* of new skills, Web sessions last tens of minutes, comprise several searches, and span many of pages. In contrast, when users want to *remind* themselves of forgotten functionality such as a function name, a single Web search and a glance at the search result summaries are often adequate. Despite programmers’ diverse Web tasks, they typically use the *same general purpose tools* to complete all of them. As a consequence, when a programmer searches for example code, he uses a Web browser that is independent of other tools in his tool chain, a search engine that has no notion of his current development context, and a code editor that assumes that all code is typed by hand. We believe that there is significant latent value in building task-specific Web search interfaces.

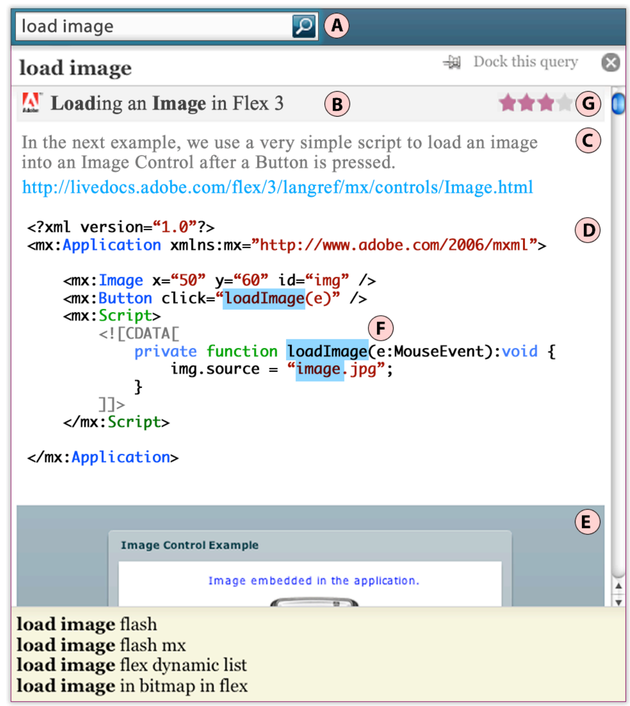

Figure 1. The Blueprint plug-in for the Adobe Flex Builder development environment helps programmers locate example code. A hotkey places a search box (A) at the programmer’s cursor position. Search results (B) are example-centric; each result contains a brief textual description (C), the example code (D), and, when possible, a running example (E). The user’s search terms are highlighted (F), facilitating rapid scanning of the result set. Blueprint allows users to rate examples (G).

We propose that giving users a *task-specific* search engine *integrated into their existing work environment* can significantly reduce the cost of searching for relevant information. Small performance improvements in routine tasks can cause qualitative behavior changes that far exceed the benefits of decreased task completion time [13]. We believe that reducing search cost through tool integration *may cause a fundamental change in the types of tasks for which people choose to use search*. These ideas are manifest in *Blueprint*, a Web search interface integrated into the Adobe Flex Builder development environment that helps users locate example code.

This paper makes two main contributions. First, it details the design and implementation of a task-specific Web search interface that helps programmers locating example code. Second, it offers an empirical comparison of this task-specific search interface to a traditional Web search interface. This empirical work comprises two parts: A comparative laboratory study with 20 participants and a query log analysis from a three-month deployment with 2024 users.

There are two key insights that enable the Blueprint search interaction, as shown in Figures 1 and 2. First, *placing the search interface inside the development environment* allows the search engine to leverage the users' context (*e.g.* programming languages and framework versions in use). This lowers the cost of constructing a good query, which improves result quality. Second, *extracting example code from Web pages and displaying it in a consistent, code-centric manner* reduces the need to click through to Web pages to find example code. This allows users to evaluate results much more rapidly than with traditional Web search interfaces, reducing the cost of selecting an optimal result.

In our empirical work, we found that Blueprint enabled laboratory study participants to search for and select example code significantly faster than with traditional Web search tools. During directed tasks in the lab, participants wrote significantly better code when using Blueprint than when using traditional Web search tools. This suggests that users are looking at a greater number of examples when using Blueprint and are therefore able to choose a better example from which to borrow. Log analysis from our deployment revealed that Blueprint's result interface dramatically reduced the need for users to click through and examine individual Web pages to find the right result. Finally, our log analysis offers several indications that users are searching differently with Blueprint than with traditional tools (*e.g.* re-finding information is significantly more common with Blueprint). This suggests that task-specific search interfaces may cause a fundamental shift in how and when individuals search the Web.

This research is inspired by prior work in two domains: tailoring Web search to specific tasks and domains, and providing support for example-centric development.

Task-Specific Search Interfaces

Prior work on tailoring search interfaces [17] has explored decision-making tasks [10, 11, 21], Web page revisitation tasks [3, 26], and, most relevant to our work, programming tasks [4, 18, 25]. The interface for Blueprint builds on recent work in providing alternative representations for search results. [3, 10, 11, 21, 26, 28]. Blueprint presents results in a templatized manner similar to Dontcheva et al.'s Web cards [10]. Displaying results from a diverse set of pages in a consistent manner allows users to rapidly browse and evaluate results.

There are research [18, 25] and commercial [1, 2] systems designed to improve search for programmers. While these search engines are *domain-specific*, they are still designed to support a broad range of tasks. Blueprint, on the other hand, is *task-specific*: it is oriented specifically towards finding example code. This introduces a trade-off: Blueprint's interface is optimized for a specific task, but loses generality. These systems are also *completely independent* of the user's development environment.

CodeTrail explores the benefits of integrating Web browsing tools and development environments by linking the Firefox browser and Eclipse IDE [12]. Blueprint goes one step further to by placing search *directly inside* the development environment. Again, this introduces a trade-off: Blueprint gives up the rich interactions available in a complete, stand-alone Web browser in favor of a more closely-coupled interaction for a specific task.

Example-Centric Development

Prior work has created tools to assist with example-centric development [16]. This work has addressed the availability of example code problem by mining code repositories [24, 27] or synthesizing example code from API specifications [20]. Blueprint is unique in that it uses regular Web pages (*e.g.* forums, blogs, and tutorials) as sources for example code. We believe using regular Web pages as sources for example code has two major benefits: First, it may provide better examples. Code written for a tutorial is likely to contain better comments and be more general purpose than code extracted from an open source repository. Second, because these pages also contain text, programmers can use natural language queries to find the code they are looking for.

The remainder of this paper proceeds as follows. We first present a scenario that describes the use of Blueprint and presents its interface. We then describe the implementation of Blueprint. Next, we detail the evaluation of Blueprint through a comparative laboratory study and a 3-month deployment. We conclude by positioning Blueprint in a design space of tools that support example-centric development.

SCENARIO: DEVELOPING WITH BLUEPRINT

Jenny is prototyping a Web application for power consumption comparison and needs to *load user data* from a server into the client application; *display feedback* while the data is being retrieved; and *visualize* the data.

First, Jenny's program needs to retrieve power-usage data from Web service that returns XML-formatted data. Although Jenny has written similar code previously, she can't remember the exact code she needs to fetch data from the Web. She *does* remember that one of the main classes involved began with “URL”. So, she types “URL” into her code and uses auto-complete to remember the “URLLoader” class. Although, she now knows the class name, Jenny still doesn’t know how to use it (Figure 2, step 1). With another hotkey Jenny brings up the Blueprint search interface, which automatically starts searching for URLLoader (step 2). Blueprint augments Jenny’s query with the language and framework version she is using, and returns appropriate examples that show how to use a URLLoader. She scans through the first few examples and sees one that creates an XML object (step 3). She selects the lines she wants to copy, presses *Enter*, and the code is pasted in her project. Blueprint augments the code with a machine- and human-readable comment that records the URL of the source and the date of copy (step 4). When Jenny opens this source file in the future, Blueprint will check this URL for changes to the source example (*e.g.,* with a bug fix), and will notify her if an update is available. Jenny runs the code in Flex’s debugger to inspect the XML data.

Finally, Jenny wants to explore different charting components to display power usage. She invokes Blueprint a third time and searches for “charting”. Jenny docks the Blueprint result window as a panel in her development environment so she can browse the results in a large, persistent view. When source pages provide a running example, Blueprint presents this example next to the source code. Eventually Jenny picks a line chart, copies the example code from the Blueprint panel into her project, and modifies it to bind the chart to the XML data. After only a few minutes her prototype is complete.

IMPLEMENTATION

Blueprint comprises a *client plug-in*, which provides the user interface for searching and browsing results, and the *Blueprint server*, which executes searches for example code. Figure 3 provides a visual system description.

Client-Side Plug-In

The Blueprint client is a plug-in for Adobe Flex Builder. Flex Builder, in turn, is a plug-in for the Eclipse Development Environment.The Blueprint client provides three main pieces of functionality. First, it provides a user interface for initiating queries and displaying results. Second, it sends contextual information (*e.g.* programming language and framework version) with each user query to the server. Third, it notifies the user when the Web origin of examples they adapted has updated (*e.g.,* when a bug is fixed). All communication between the client and sever occurs over HTTP using the JSON data format.

Blueprint’s query and search results interface is implemented using HTML and JavaScript that are rendered by SWT browser widgets. Search results are rendered sequentially in a list below the query box. Each search result includes the source Web page title, a hyperlink to the source Web page, English description of the example, and the code example. All examples include syntax highlighting (produced by the Pygments library), and users can navigate through examples using the Tab key and copy/paste selec-

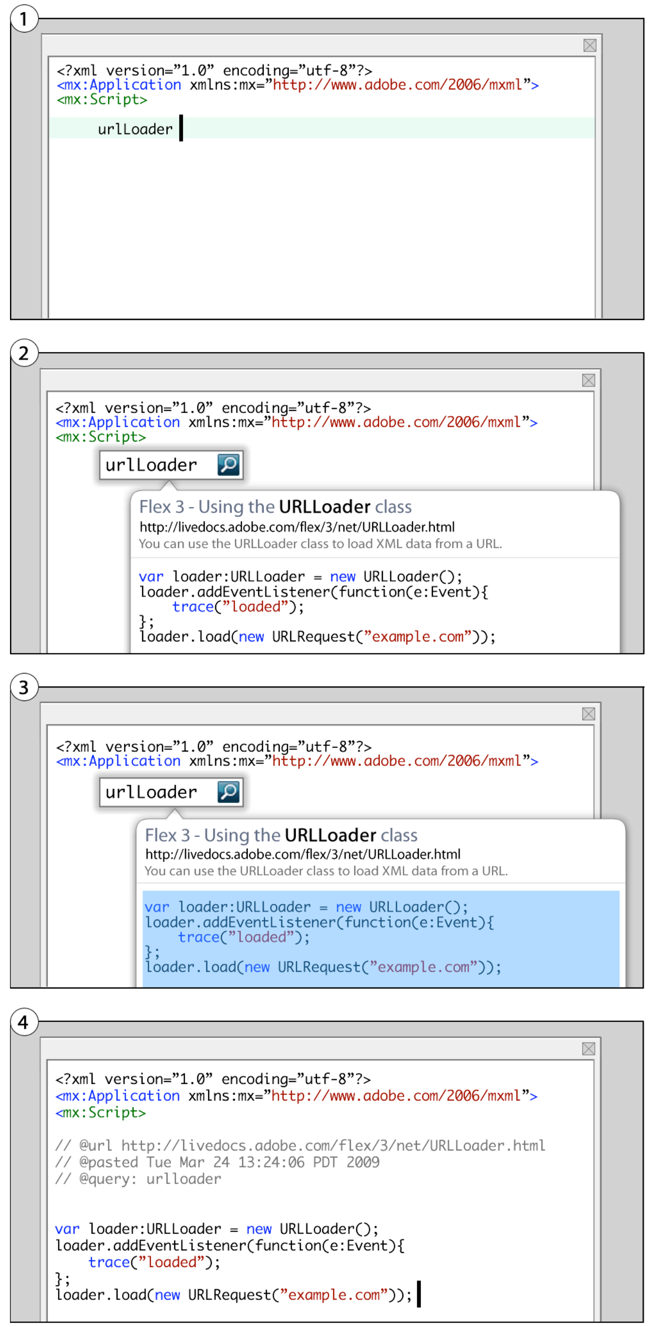

Figure 2. Example-centric programming with Blueprint. The user presses a hotkey to initiate a search; a search box appears at the cursor location (1). Searches are performed interactively as the user types; example code and running examples (when present) are shown immediately (2). The user browses examples with the keyboard or mouse, and presses *Enter* to paste an example into her project (3). Blueprint automatically adds a comment containing metadata that links the example to its source (4).

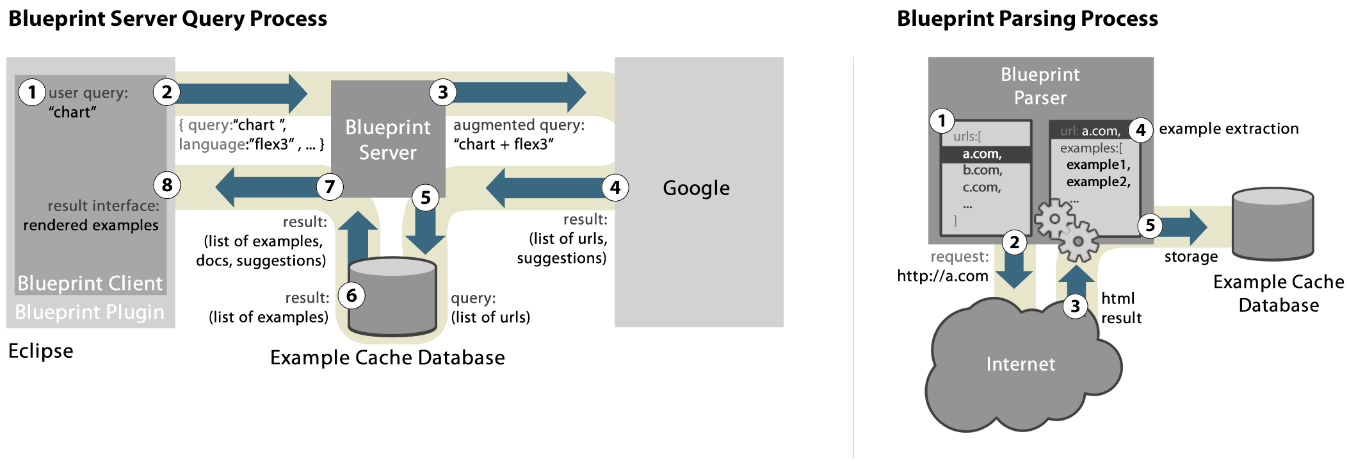

Figure 3. Architecture of the Blueprint server. The process of servicing a user's query is shown on the left; the background task of parsing Web pages to extract examples is shown on the right.

tions by pressing enter. Users can rate examples and dock the Blueprint floating window as an Eclipse panel. Blueprint also allows users to follow hyperlinks to view search results in context, and maintains a browsing and search history.

When a user pastes example code into their project, Blueprint inserts a Javadoc-like comment at its beginning. This comment tags the example code with its *URL* source, the insertion *date and time*, and a *unique numerical identifier*. This metadata is both human- and machine-readable. Blueprint searches for these example comments each time a file is opened. For each comment, it queries the Blueprint server to check if the original example has been modified since it was copied.

Blueprint Server

The Blueprint server executes queries for example code and returns examples to the client. To maximize speed, breadth, and ranking quality, the server leverages a general-purpose search engine. Blueprint uses the Adobe Community Help search APIs, a Google Custom Search engine. This search engine indexes Adobe product-specific content from across the Web. When the Blueprint server receives a query, it first augments the query with the user's context (e.g. programming language and framework version), which is sent along with the query by the client. Then the server sends the new augmented query to the search engine, which returns a set of URLs. Since Blueprint users are interested in code *examples* and *not Web pages*, Blueprint process the Web pages returned by the search engine to extract source code examples.

Since processing each page requires on average 10 seconds (8 seconds to retrieve the page, 2 second to extract examples), we preprocess pages and cache them. Unfortunately, Blueprint does not have access to the global list of pages indexed by the search engine, and when the search engine returns URLs that are not in the Blueprint cache, the URLs are added to the cache by a background process. Code example from those URLs are returned in future queries.

Before deploying Blueprint, we pre-populated the cache with approximately 50,000 URLs obtained from search engine query logs. To keep the cache current, Blueprint crawls the URLs in the cache as a background process. Since pages containing examples are relatively static, the Blueprint prototype re-crawls them weekly.

Leveraging an existing commercial search engine to produce a candidate result set has a number of advantages over building a complete search engine from scratch (*e.g.* [18, 25]). First, it is substantially more resource-efficient to implement, as keeping a document collection up to date is expensive. Second, generating high-quality search results from natural-language queries is a hard problem and a highly-optimized commercial search engine is likely to produce better results than a prototype search engine with a restricted domain. Finally, a general-purpose search engine surfaces examples from tutorials, blogs, and API pages. Examples found on such pages are more likely to be instructive than examples extracted from large source code repositories.

Extracting Example Code and Descriptions

To extract source code from Web pages, Blueprint segments the page and classifies each segment as source code or other type of content. First, Blueprint uses the BeautifulSoup library [23] to transform HTML into proper XHTML, and then it divides the resulting hierarchical XHTML document into independent segments by examining block-level elements. Blueprint uses 31 tags to define blocks; the most common are: P, H1, DIV, and PRE. It also extracts SCRIPT and OBJECT blocks as block-level elements, because running examples are usually contained within these tags. To find block-level elements, Blueprint performs a depth-first traversal of the document. When it reaches a leaf element, it backtracks to the nearest block-level ancestor and creates a segment. If the root of the tree is reached before finding a block-level element, the element immediately below the root is extracted as a segment. This algorithm keeps segments ordered exactly as they were in the original file. Finally, to more easily and reliably determine whether a segment contains code, Blueprint renders each segment to plain text using w3m, a text-based Web browser. This rendering allows for classification of code based on its appearance to a user on a Web page and not based on its HTML structure.

Blueprint stores the HTML and plain text versions of all segments in a database. On average, a Web page in our dataset contains 161 segments. However, 69% of these are less than 50 characters long (these are primarily created by navigational elements). Although this algorithm leads to a large number of non-source code segments, it correctly parses blocks of example code into single segments, which enables our classifiers to prune non-source code segments.

One limitation of this extraction algorithm is that it assumes code examples on Web pages are independent and so it does not handle Web pages that provide several related code examples that should be considered in concert, such as tutorials that list several steps or offer several complementary alternatives.

Classifying example code

Given a set of clean, separate segments, the most straightforward approach to classifying them as source code, is to use a programming language parser and label segments that parse correctly as code. For Blueprint, this would require ActionScript and MXML parsers, because they are the two languages used by Adobe Flex. In practice, this approach yields many false negatives: segments that contain code but are not labeled as such. For example, code with line numbers or a typo will cause parsing to fail.

An alternate approach is to specify heuristics based on features unique to code, such as curly braces, frequent use of language keywords, and lines that end with semi-colons [18]. This approach produces many fewer false negatives, but introduces false positives, such as paragraphs of text that discuss code. Such paragraphs usually describe other source code found on the page and are not useful on their own.

We built the Blueprint heuristic classifiers by identifying language keywords and unique syntax features. The MXML classifier looks for the mxml namespace identifier (http://www.adobe.com/2006/mxml) and the mxml namespace tags (<mx:). It counts the number lines that start with an angle bracket and checks for closing tags on the last line. The ActionScript classifier looks for camel case words, periods and parentheses in the middle of words, single curly braces on lines, and semicolons at the end of lines. The classifiers compute a score for each segment and classify all segments with a score above a threshold (MXML: 3, AS: 10) and with a minimum number of lines (MXML: 2, AS: 6) as source code.

In addition to computing a segment score, the classifiers also include a number of requirements. For example, the MXML classifier requires that the segment begins with an open bracket (<) and ends with a close bracket (>). The ActionScript classifier requires at least one equal sign (=) or 'function' keyword and a line to word ratio that is greater than 0.15. This line to word ratio is useful in pruning text paragraphs that include code, as text paragraphs have many more words on each line than source code segments.

To remove buggy code that appears in forums and blog post comments, we ignore all segments that follow a comment block, where a comment block is a block that includes the word "comment" and all Web pages that include 'group' or 'forum' in the URL.

We computed precision (MXML: 84%, AS: 91%) and recall (MXML: 90%, AS: 86%) on 40 randomly sampled Web pages from a corpus of the 2000 most frequently visited Web pages from the Adobe Community Help Search Web site. We compared the examples extracted by Blueprint to the examples manually extracted by two researchers. Precision was mainly affected by classifying source examples in other languages to MXML and ActionScript, such as HTML, Javascript, and Coldfusion. Recall differed among types of Web pages. API reference Web pages, which are often produced automatically, were much easier to parse than tutorial Web pages, which vary greatly in the types of examples they show. The current Blueprint cache includes 59,424 examples from 21,665 unique Web pages.

Extracting text and running examples

In addition to extracting source code, Blueprint extracts English descriptions and, where possible, running examples for each code segment. Informal inspection of pages containing example code found that the text immediately preceding an example almost always described the example, and running examples almost always occurred after the example code.

To build descriptions, Blueprint iteratively joins the segments immediately preceding the code until any of three conditions is met: 1.) we encounter another code segment, 2.) we encounter a segment indicative of a break in content (those generated by DIV, HR, or heading tags), or 3.) we reach a length threshold (currently 250 words). Using this strategy the English we extract is the correct example description roughly 83% of the time.

To find running examples, Blueprint analyzes the k segments following a code example. Because we are concerned with Flex, all examples occur as Flash SWF files. We search for references to SWF files in OBJECT and SCRIPT tags. In practice, we have found k=3 works best; larger values resulted in erroneous content, such as Flash-based advertisements.

Keeping track of changes to examples

Each time a page is crawled, Blueprint checks for updates to the examples (*e.g.,* bug fixes). It performs an exhaustive, pairwise comparison of examples on the new and old pages using the *diff* tool. Pages typically contain fewer than ten examples. If an example on the new and old pages matches exactly, they are deemed the same. If a new example has more than two-thirds of its lines in common with an old example, it is recorded as changed. Otherwise, the new example is deemed new. The database stores each example with a timestamp, and keeps all previous versions. Timestamping examples helps Blueprint notify users when an example changes.

EVALUATION: STUDYING BLUEPRINT IN THE LAB

We conducted a comparative laboratory study with 20 participants followed by and a 3-month large-scale deployment with 2024 users to better understand how Blueprint affects the example-centric development process.

Comparing Speed and Code Quality

The comparative laboratory study evaluated three hypotheses:

H1: Programmers using Blueprint will complete directed tasks more quickly than those who do not because they will find example code faster and bring it into their project sooner.

H2: Code produced by programmers using Blueprint will have the same or higher quality as code written by example modification using traditional means.

H3: Programmers who use Blueprint produce better designs on an exploratory design task than those using a Web browser for code search.

Method

Twenty professional programmers participated in this study. We recruited them through an internal company mailing list and compensated them with a $15 gift card in exchange. Participants had an average of 11.3 years of professional experience. Fourteen reported at least one year of programming experience with Flex; twelve reported spending at least 25 hours a week programming in Flex.

Participants were given an off-the-shelf installation of Flex Builder, pre-loaded with three project files. Participants in the control condition were provided with the Firefox Web browser; they were asked to use the Adobe Community Help Search engine to look for example code. Participants in the treatment condition were provided with Blueprint to search for code samples; they were not allowed to use a Web browser.

Participants were asked to complete a *tutorial*, a *directed task*, and an *exploratory task*. Participants were told that they would be timed and that they should approach all tasks as though they are prototyping and not writing production-level code. Participants began each task with a project file that included a running application, and they were asked to add additional functionality.

For the *tutorial* task, the sample application contained an HTML browsing component and three buttons that navigate the browser to three different Web sites. Participants received a written tutorial that guided them through adding fade effects to the buttons and adding a busy cursor. In the control condition, the participants were asked to use the Web browser to find sample code for both modifications. The tutorial described which search result would be best to follow and which lines of code to add to the sample application. In the treatment condition, the participants were asked to use Blueprint to find code samples.

For the *directed programming* task, the participants were instructed to use the *URLLoader* class to retrieve text from a URL and place it in a text box. They were told that they should complete the task as quickly as possible. In addition, the participants were told that the person to complete the task fastest would receive an additional gift card as a prize. Participants were given 10 minutes to complete this task.

Figure 4. Comparative laboratory study results. Each graph shows the relative rankings of participants. Participants who used Blueprint are shown as filled squares, those who used Community Help are shown as open squares.

For the *exploratory programming* task, participants were instructed to use Flex Charting Components to visualize an array of provided data. The participants were instructed to make the best possible visualization. They were told that the results would be judged by an external designer and the best visualization would win an extra gift card. Participants were given 15 minutes to complete this task.

To conclude the study, we asked the participants a few questions about their experience with the browsing and searching interface.

Results

Directed Task

Nine out of ten Blueprint participants and eight out of ten control participants completed the directed task. Because not all participants completed the task and completion time may not be normally distributed, we report all significance tests using rank-based non-parametric statistical methods (Wilcoxon-Mann-Whitney test for rank sum difference and Spearman rank correlation).

We ranked the participants by the time until they pasted the first example. See Figure 4. Participants using Blueprint pasted code for the first time after an average of 57 seconds, versus 121 seconds for the control group. The rank-order difference in time to first paste was significant ($p < 0.01$). Among finishers, those using Blueprint finished after an average of 346 seconds, compared to 479 seconds for the control. The rank-order difference for all participants in task completion time was not significant (p=0.14). Participants' first paste time correlates strongly with task completion time (r_s=0.52, p=0.01). This suggests that lowering the time required to search for, selecting and copying examples will speed development.

A professional software engineer external to the project rank-ordered the participants' code. He judged quality by whether the code met the specifications, whether it included error handling, whether it contained extraneous statements, and overall style. Participants using Blueprint produced significantly *higher-rated* code (p=0.02). We hypothesize this is because the example-centric result view in Blueprint makes it more likely that users will choose a good example to start from. When searching for "URLLoader" using the Adobe Community Help search engine, the first result contains the best code. However, this result's *snippet* did not convey that the page was likely to contain sample code. For this reason, we speculate that some control participants overlooked it.

Exploratory Task

A professional designer rank-ordered the participants' charts. To judge chart quality, he considered the appropriateness of chart type, whether or not all data was visualized, and aesthetics of the chart. The sum of ranks was smaller for participants using Blueprint (94 vs. 116), but this result was not significant (p=0.21). While a larger study may have found significance with the current implementation of Blueprint, we believe improvements to Blueprint's interface (described below) would make Blueprint much more useful in exploratory tasks.

Areas for Improvement

When asked "How likely would you be to install and use Blueprint in its current form?" participants responses averaged 5.1 on a 7-point Likert scale (1 = "not at all likely", 7 = "extremely likely"). Participants also provided several suggestions for improvement.

The most common requests were for greater control over result ranking. Two users suggested that they should be able to rate (and thus affect the ranking of) examples. Three users expressed interest in being able to filter results on certain properties such as whether result has a running example, the type of page that the result was taken from (blog, tutorial, API documentation, etc.), and the presence of comments in the example. Three participants requested greater integration between Blueprint and other sources of data. For example, one participant suggested that all class names appearing in examples be linked to their API page. Finally, three participants requested maintaining a search history; one also suggested a browseable and searchable history of examples used. We implemented the first two suggestions before the field deployment. The third remains future work.

Discussion

In addition to the participants' explicit suggestions, we identified a number of shortcomings as we observed participants working. It is currently difficult to compare multiple examples using Blueprint. Typically, only one example fits on the screen at a time. To show more examples simultaneously, one could use code-collapsing techniques to reduce each example's length. Additionally, Blueprint could show all running examples from a result set in parallel. Finally, visual differencing tools might help users compare two examples.

We assumed that users would only invoke Blueprint once per task. Thus, each time Blueprint is invoked, the search box and result area would be empty. Instead, we observed that users invoked Blueprint multiple times for a single task (*e.g.* when a task required several blocks of code to be copied to disparate locations). Results should be persistent, but it should be easier to clear the search box: when re-invoking Blueprint, the terms should be pre-selected so that typing replaces them.

LONGITUDINAL STUDY: DEPLOYMENT TO 2,024 USERS

To better understand how Blueprint would affect the workflow of real-world programmers, we conducted a three-month deployment on the Adobe Labs Web site. Over the course of the deployment, we performed bug fixes and minor design improvements (often based on feedback through the Web forum); the main interaction model remained constant throughout the study.

At the completion of the study, we conducted 30-minute interviews with four active Blueprint users to understand how they integrated Blueprint in their workflows. Based on the interviews, we formed three hypotheses, which we tested with the Blueprint usage logs. After evaluating these hypotheses, we performed further exploratory analysis of the logs. This additional analysis provided high-level insight about current use that we believe will help guide future work in creating task-specific search interfaces.

Insights from Interviewing Active Users

Our interviews with active users uncovered three broad insights about the Blueprint interface. To understand if these insights generalize, we distilled each insight into a testable hypothesis. The insights and hypotheses are presented here; the results of testing them are presented in the following section.

The benefits of consistent, example-centric results outweigh the drawbacks of missing context.

A consistent view of results makes scanning the result set more efficient. However, in general, removing content from its context may make understanding the content more difficult. None of our interviewees found lack of context to be a problem when using Blueprint. One interviewee walked us through his strategy for finding the right result: "Highlighting [of the search term in the code] is the key. I scroll through the results quickly, looking for my search term. When I find code that has it, I can understand the code much faster than I could English." We hypothesize that examining code to determine if a result is relevant has a smaller gulf of evaluation [19] than examining English. Presenting results in a constant manner makes this process efficient.

When users desire additional context for a Blueprint result, they can click through to the original source Web page. This Web page opens in the same window where Blueprint results are displayed. If additional context is rarely necessary, we expect a low click-through rate.

H1: *Blueprint will have a significantly lower click-through rate than seen in a standard search engine.*

Blueprint is symbiotic with existing IDE features; they each make the other more useful.

Three interviewees reported using Blueprint as an "extension" to auto-complete. They use auto-complete as an index into a particular object's functionality, and then use Blueprint to quickly understand how that functionality works. This suggests that embedding search into the development environment creates a symbiotic relationship with other features. Here, auto-complete becomes more useful because further explanation of the auto-complete results is one keystroke away. We believe that this symbiotic relationship is another example of how integrating task-specific search into a user's existing tools can lower search costs.

Programmers routinely search with code terms when using standard search engines [6]. However, when these search terms are typed by hand, they frequently contain formatting inconsistencies (*e.g.* method names used as search terms are typed in all lowercase instead of camelCase). By contrast, when search terms come *directly from* a user's code (*e.g.* generated by output from auto-complete), the search terms will be correctly formatted. If Blueprint is being used in a symbiotic manner with other code editing tools, we expect to see a large number of correctly formatted queries.

H2: *Blueprint search terms will contain correctly formatted code more often than search terms used with a standard search engine.*

Blueprint is used heavily for clarifying existing knowledge and reminding of forgotten details.

One interviewee stated that, using Blueprint, he could find what he needed "60 to 80 percent of the time without having to go to API docs." He felt that Blueprint fell in the "mid-space between needing to jump down into API docs when you don't know what you're doing at all and not needing help because you know exactly what you are doing." Other interviewees echoed this sentiment. In general, they felt that Blueprint was most useful when they had some knowledge about how to complete the task at hand, but needed a piece of clarifying information.

In general, understanding a user's search goal from query logs alone is not feasible—there is simply not enough contextual information available [14]. However, if uses of Blueprint tend more toward reminding and clarifying existing knowledge than learning new skills, we expect that users will more commonly repeat queries they have performed in the past.

H3: *Users of Blueprint are more likely to repeat queries across sessions than users of a standard search engine.*

Methodology

To evaluate these hypotheses, one needs a comparison point. Adobe's Community Help search engine presents a standard Web search interface that is used by thousands of Flex programmers. Furthermore, Community Help uses the same Google Custom Search Engine that is part of Blueprint. In short, Blueprint and Community Help differ in their interaction model, but are similar in search algorithm, result domain, and user base.

We randomly selected 5% of users who used the Community Help search engine over the same period as the Blueprint deployment. We analyzed all logs for these users. In both datasets, queries for individual users were grouped into *sessions*. A session was defined as a sequence of events from the same user with no gaps longer than six minutes. (This grouping technique is common in query log analysis, e.g. [6].) Common "accidental" searches were removed (*e.g.*, empty or single-character searches, and identical searches occurring in rapid succession) in both datasets.

We used the z-test for determining statistical significance of differences in means and the chi-square test for determining differences in rates. Unless otherwise noted, all differences are statistically significant at $p < 0.01$.

Results

Blueprint was used by 2024 individuals during the 82 day deployment, with an average of 25 new installations per day. Users made a total of 17012 queries, or an average of 8.4 queries per user. The 100 most active users made 1888 of these queries, or 18.8 queries per user.

The Community Help query logs used for comparison comprised 13283 users performing 26036 queries, an average of 2.0 queries per user.

H1: *Blueprint will have a significantly lower click-through rate than seen in a standard search engine*

Blueprint users clicked through to source pages significantly less than Community Help users (μ = 0.38 versus 1.32). To be conservative: the mean of 0.38 for Blueprint is an over-estimate. For technical reasons owing to the many permutations of platform, browser, and IDE versions, click-throughs were not logged for some users. For this reason, this analysis discarded all users with zero click-throughs.

H2: *Blueprint search terms will contain correctly formatted code more often than search terms used with a standard search engine.*

To test this hypothesis, we used the occurrence of camelCase words as a proxy for code terms. The Flex framework's coding conventions use camelCase words for both class and method names, and camelCase rarely occurs in English words.

Significantly more Blueprint searches contained camelCase than Community Help: 49.6% (8438 of 17012) versus 16.2% (4218 of 26036). The large number of camelCase words in Blueprint searches indicates that many searches are being generated directly from users' code. This suggests that, as hypothesized, Blueprint is being used in a symbiotic way with other IDE features. The large number of camelCase queries in Blueprint searches also indicates that the majority of searches use precise code terms. This suggests that Blueprint is being used heavily for clarification and reminding, where the user has the knowledge necessary to select precise search terms.

H3: *Users of Blueprint are more likely to repeat queries across sessions than users of a standard search engine.*

Significantly more Blueprint search sessions contained queries that had been issued by the same user in an earlier session than for Community Help: 12.2% (962 of 7888) versus 7.8% (1601 of 20522).

Exploratory Analysis

To better understand how Blueprint was used, we performed additional exploratory analysis of the usage logs. We present our most interesting findings below.

Query refinement is common

Across all sessions, Blueprint users had a significantly higher number of refinements per session than Community Help users (μ = 1.8 versus 1.2).. It is difficult to understand from a quantitative perspective *why* Blueprint users are refining their queries more often. One possible explanation is that query refinement is more common because Blueprint

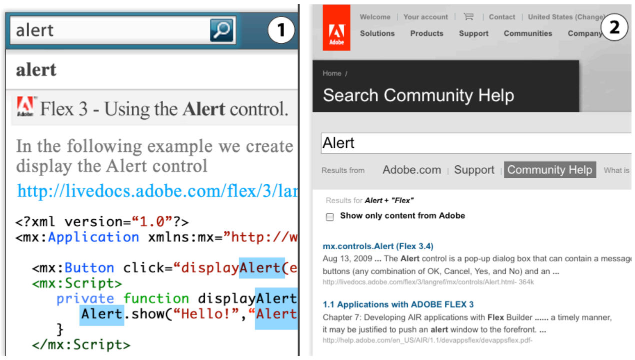

Figure 5. Comparison of Blueprint (left) and Commmunity Help (right) search result interfaces for the query "Alert". The desired information is immediately available in Blueprint; Community Help users must click the first result and scroll part way down the page to find the same information.

queries are cheaper, as users do not need to leave their current environment. Similarly, it is less costly to attempt multiple queries with Blueprint because the quality of results can be determined without clicking on result links.

Using Blueprint as a resource to write code by hand is common.

76% of sessions did not contain a copy-and-paste event. There are two possible reasons for this high number: First, as our interviewees reported, we believe Blueprint is commonly used to confirm that the user is on the right path – if they are, they have nothing to copy. Second, sometimes Blueprint's results aren't useful. (For technical reasons, copy-and-paste events were not logged on some platforms. The statistic presented here is only calculated amongst users were we could log this event. In this data set, there were 858 sessions that contained copy-and-paste events out of a total of 3572 sessions.)

People search for similar things using Blueprint and Community Help, but the frequencies are different.

We examined the most common queries for Blueprint and Community Help and found that there was a large amount of overlap between the two sets: 10 common terms appeared in the top 20 queries of both sets. The relative frequencies, however, differed between sets. As one example, the query "Alert" was significantly more frequent in Blueprint than Community Help. It occurred 2.2 times more frequent, ranking 8^{th} versus 34^{th}.

The initial result views for search "Alert" for both Blueprint and Community Help are shown in Figure 5. In the case of this particular search, we believe the difference in frequency is explained by the granularity of the task the user is completing. Namely, this task is small. When a user searches for "Alert," he is likely seeking the one line of code necessary to display a pop-up alert window. In Blueprint, the desired line is immediately visible and highlighted; in Community Help, the user must click on the first result and scroll part way down the resulting page to find the code. Alerts are often used for debugging, where there are reasonable—but less optimal—alternative approaches (*e.g.* "trace" statements). It may be the case that Blueprint's lowered search cost changes user behavior. Users who do not have Blueprint more frequently settle for sub-optimal approaches because of the relatively higher cost of taking the optimal approach.

Both interface modalities are important

Users can interact with blueprint either as a pop-up window or inside a docked panel. Among all users, 59% of sessions used only the pop-up interface, 9% used only the docked interface, and 32% used both. This suggests that providing both interfaces is important. Furthermore the fact that users frequently switched between interfaces mid-session suggests that some tasks are more appropriate for a particular interface.

DESIGN SPACE

This section discusses the important decisions made in Blueprint's design, and positions them in a space of alternative designs (see Figure 6). Positioning Blueprint within this space helps structure a discussion of Blueprint's limitations and suggests fruitful areas for future work.

Task: At a high level, programming comprises: planning and design; implementation; and testing and debugging. Blueprint helps programmers find code that implements desired functionality. Other tasks could (and do) benefit from Web search [25], but aren't right for Blueprint. For example, to decipher a cryptic error message, one may want to use program output as the search query.

Expertise: Programmers vary in expertise with the tools they use (*e.g.* languages and libraries), and their tasks (*e.g.* implementing a piece of functionality). Because Blueprint presents code-centric results, programmers must have the expertise required to evaluate whether a result is appropriate. With this expertise, code offers low cognitive thresholds for executing searches and evaluating the utility of results.

Time scale: The cost of a short task is dominated by activation energy. The cost of a long task is dominated by feature richness. We designed Blueprint to make small tasks faster by directly integrating search into the code editor. This removes the activation barrier of invoking a separate tool. While Blueprint can be docked to be persistent, for longer information tasks, the advantages of a richer browser will dominate the time savings of direct integration.

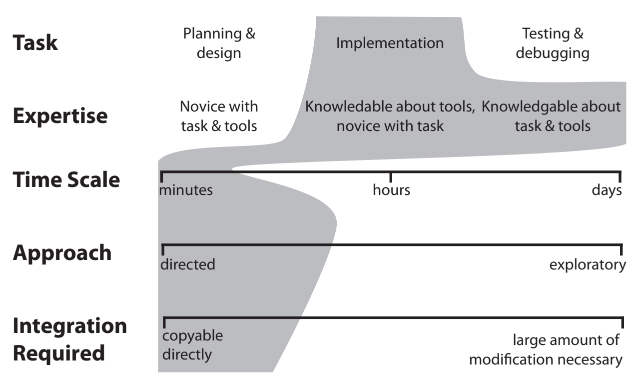

Figure 6. Design space of tools to aid programmers' Web use. Blueprint is designed to address the portion of the space shown with a shaded background.

Approach: Sometimes, one is looking for a particular piece of information. Other times, one's goals are more exploratory. The Blueprint prototype is designed for directed tasks: a well-specified query can efficiently retrieve a desired result. It is possible to use Blueprint for some more exploratory tasks, such as browsing different types of charts. However, doing so was not a primary use case. Perhaps its most notable shortcoming for exploratory tasks is that only one result is shown at a time; results cannot be simultaneously compared.

Integration Required: Some examples can be directly copied. Others require significant modification to fit the current context. Because Blueprint inserts example code directly into the user's project, it provides the most benefit when example code requires little modification. When a piece of code is deeply intertwined with a larger context, it may be easier to create the new instance while referring to the example rather than copying it.

CONCLUSION

To support programming by example modification, this paper introduced a user interface for accessing online example code from *within the development environment*. It discussed the Blueprint client interface, which displays search results in an *example-centric manner*. The Blueprint server introduced a lightweight architecture for using a general-purpose search engine to create code-specific search results that include written descriptions and running examples. In evaluating Blueprint, we found that it enabled users to search for and select example code significantly faster than with traditional Web search tools. Log analysis from a large-scale deployment with 2,024 users suggested that task-specific search interfaces may cause a fundamental shift in how and when individuals search the Web.

REFERENCES

1 *Google Code Search*. http://code.google.com
2 *Krugle*. http://www.krugle.com
3 Adar, E., M. Dontcheva, J. Fogarty, and D. S. Weld. Zoetrope: Interacting with the Ephemeral Web. In *Proceedings of UIST: ACM Symposium on User Interface Software and Technology*. pp. 239-48, 2008.
4 Bajracharya, S., T. Ngo*, et al.* Sourcerer: A Search Engine for Open Source Code Supporting Structure-Based Search. In *Companion to OOPSLA: ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications*. pp. 681-82, 2006.
5 Brandt, J., P. J. Guo, J. Lewenstein, M. Dontcheva, and S. R. Klemmer. Opportunistic Programming: Writing Code to Prototype, Ideate, and Discover, *IEEE Software*, vol. 26(5): pp. 18-24, 2009.
6 Brandt, J., P. J. Guo, J. Lewenstein, M. Dontcheva, and S. R. Klemmer. Two Studies of Opportunistic Programming: Interleaving Web Foraging, Learning, and Writing Code. In *Proceedings of CHI: ACM Conference on Human Factors in Computing Systems*, 2009.
7 Brooks, F. P., *The Mythical Man-Month: Essays on Software Engineering.* Addison-Wesley. 1995.
8 Clarke, S. What is an End-User Software Engineer? In *End-User Software Engineering Dagstuhl Seminar*, 2007.
9 deHaan, P., *Flex Examples*. http://blog.flexexamples.com/
10 Dontcheva, M., S. M. Drucker, D. Salesin, and M. F. Cohen. Relations, Cards, and Search Templates: User-Guided Web Data Integration and Layout. In *Proceedings of UIST: ACM Symposium on User Interface Software and Technology*. pp. 61-70, 2007.
11 Dontcheva, M., S. M. Drucker, G. Wade, D. Salesin, and M. F. Cohen. Summarizing Personal Web Browsing Sessions. In *Proceedings of UIST: ACM Symposium on User Interface Software and Technology*. pp. 115-24, 2006.
12 Goldman, M. and R. C. Miller. Codetrail: Connecting Source Code and Web Resources. In *Proceedings of VL/HCC: IEEE Symposium on Visual Languages and Human-Centric Computing*. pp. 65-72, 2008.
13 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.
14 Grimes, C., D. Tang, and D. M. Russell. Query Logs Alone are Not Enough. In *Workshop on Query Log Analysis at WWW 2007*.
15 Hartmann, B., S. Doorley, and S. R. Klemmer. Hacking, Mashing, Gluing: Understanding Opportunistic Design, *IEEE Pervasive Computing*, vol. 7(3): pp. 46-54, 2008.
16 Hartmann, B., L. Wu, K. Collins, and S. R. Klemmer. Programming by a Sample: Rapidly Creating Web Applications with d.mix. In *Proceedings of UIST: ACM Symposium on User Interface Software and Technology*. pp. 241-50, 2007.
17 Hearst, M. A., *Search User Interfaces.* Cambridge University Press. 2009.
18 Hoffmann, R., J. Fogarty, and D. S. Weld. Assieme: Finding and Leveraging Implicit References in a Web Search Interface for Programmers. In *Proceedings of UIST: ACM Symposium on User Interface Software and Technology*. pp. 13-22, 2007.
19 Hutchins, E. L., J. D. Hollan, and D. A. Norman. Direct Manipulation Interfaces*. Human-Computer Interaction* **1**(4). pp. 311-38, 1985.
20 Mandelin, D., L. Xu, R. Bodík, and D. Kimelman. Jungloid Mining: Helping to Navigate the API Jungle. In *Proceedings of PLDI: ACM SIGPLAN Conference on Programming Language Design and Implementation*. pp. 48-61, 2005.
21 Medynskiy, Y., M. Dontcheva, et al. Exploring Websites through Contextual Facets. In *Proceedings of CHI: ACM Conference on Human Factors in Computing Systems*, 2009.
22 Pirolli, P. L. T., *Information Foraging Theory.* Oxford University Press. 2007.
23 Richardson, L., *Beautiful Soup*. http://www.crummy.com/software/BeautifulSoup
24 Sahavechaphan, N. and K. Claypool. XSnippet: Mining for Sample Code. In *Proceedings of OOPSLA: ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications*. pp. 413-30, 2006.
25 Stylos, J. and B. A. Myers. Mica: A Web-Search Tool for Finding API Components and Examples. In *Proceedings of VL/HCC: IEEE Symposium on Visual Languages and Human-Centric Computing*. pp. 195-202, 2006.
26 Teevan, J., E. Cutrell*, et al.* Visual Snippets: Summarizing Web Pages for Search and Revisitation. In *Proceedings of CHI: ACM Conference on Human Factors in Computing Systems*, 2009.
27 Thummalapenta, S. and T. Xie. PARSEweb: A Programmer Assistant for Reusing Open Source Code on the Web. In *Proceedings of ASE: IEEE/ACM International Conference on Automated Software Engineering*. pp. 204-13, 2007.
28 Woodruff, A., A. Faulring, R. Rosenholtz, J. Morrsion, and P. Pirolli. Using Thumbnails to Search the Web. In *Proceedings of CHI: ACM Conference on Human Factors in Computing Systems*. pp. 198-205, 2001.