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 changed the cost structure of programming by example modification. However, current search tools are wholly separate from editing tools. What benefits might be realized by integrating them? This paper describes the design, implementation, and evaluation of Blueprint, a tool that integrates Web search into the Adobe Flex Builder development environment. Blueprint *automatically augments queries with code context*, presents an *example-centric view of search results*, and retains a *link between copied code and its source*. This paper introduces a technique for retrieving relevant example code, descriptions, and running examples for a user's query. A between-subjects study found that Blueprint enables participants to search for and select example code significantly faster than with a standard Web browser.

ACM Classification: H5.2 [Information interfaces and presentation]: User Interfaces—*prototyping*. D.2.2 [Software engineering]: Design Tools and Techniques—*user interfaces*.

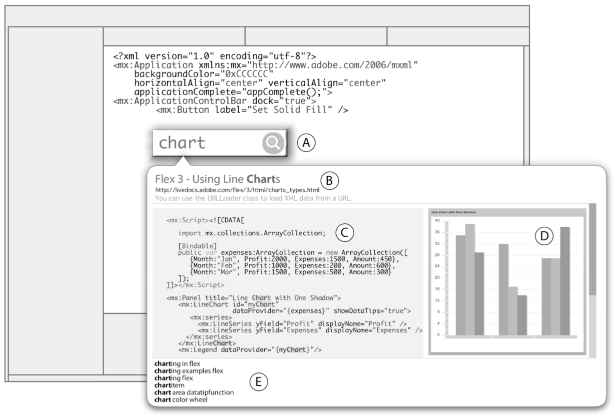
General terms: Design, Human Factors

Keywords: Example-centric development, opportunistic programming, Web search, prototyping

INTRODUCTION

Programmers routinely face the "build or borrow" question [1]: should they implement a piece of functionality from scratch, or locate and adapt relevant existing code? Web search is fundamentally changing the cost structure of this question [2]. It is now possible to quickly locate example code that implements nearly any piece of routine functionality [3]. This enables programmers to opportunistically build applications by searching for, modifying, and combining short blocks of example code taken from the Web [4-6].

In 1993, Nardi suggested that "programming by example modification" holds significant latent value. An open question at the time was "how users will find appropriate example code; for any practical application of example modification, many libraries of example code will have to be available and the information access problems will be significant." [7]. Sixteen years later, *finding* appropriate code has become easier: there are many Web sites dedicated to example sharing (*e.g.* the Flex Examples Blog [8]), online open-source code repositories (*e.g.* Google Code [9]), and search interfaces for programmers [10, 11]. However, these *search tools* are still wholly separate from *editing tools*. Current development environments provide little support for example-centric development. Instead, they tacitly assume that programming begins *tabula rasa* and that code is either written by the programmer or imported as a library module.



```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    backgroundColor="0xCCCCCC"
    horizontalAlign="center" verticalAlign="center"
    applicationComplete="appComplete();">
<mx:ApplicationControlBar dock="true">
        <mx:Button label="Set Solid Fill" />
```

chart
A
Flex 3 - Using Line Charts
B
http://livedocs.adobe.com/flex/3/html/charts_types.html
You can use the URLLoader class to load XML data from a URL.
C
D
Line Chart with One Shadow
E

```
<mx:Script><![CDATA[

   import mx.collections.ArrayCollection;

   [Bindable]
   public var expenses:ArrayCollection = new ArrayCollection([
      {Month:"Jan", Profit:2000, Expenses:1500, Amount:450},
      {Month:"Feb", Profit:1000, Expenses:200, Amount:600},
      {Month:"Mar", Profit:1500, Expenses:500, Amount:300}
   ]);
]]></mx:Script>

<mx:Panel title="Line Chart with One Shadow">
   <mx:LineChart id="myChart"
                 dataProvider="{expenses}" showDataTips="true">
      <mx:series>
        <mx:LineSeries yField="Profit" displayName="Profit" />
        <mx:LineSeries yField="Expenses" displayName="Expenses" />
     </mx:series>
   </mx:LineChart>
   <mx:Legend dataProvider="{myChart}"/>
```

charting in flex
charting examples flex
charting flex
chartitem
chart area datatipfunction
chart color wheel

Figure 1. The Blueprint plug-in for the Adobe Flex Builder development environment helps programmers engage in example-centric development. A hotkey places a search box (A) at the programmer's cursor position. Search results are example-centric; each result contains a brief textual description (B), the example code (C), and often a running example (D). Blueprint also provides additional search suggestions (E).

Several difficulties arise from separate tools for editing and search. First, the important link between borrowed code and its source is lost. The programmer may not realize the code was borrowed from an online source; this can be valuable when debugging or modifying code. If they do know it was borrowed, but not the URL, they may have difficulty re-finding it if they would like to verify attributes or view additional code and commentary. Second, if the source example is later updated (*e.g.* to fix a bug), the programmer will never know. Finally, to obtain relevant search results, programmers must manually specify contextual constraints in their query, such as languages and frameworks used.

We hypothesize that there is significant value in integrating Web search with a code editor. More specifically, this paper proposes that *automatically augmenting queries with code context* and presenting an *example-centric view of search results* increases the speed, quality, and ease of programming by example modification. We introduce *Blue-*

print, an extension to the Adobe Flex Builder development environment that manifests these ideas (see Figure 1). This paper makes two contributions.

First, it introduces a user interface that integrates searching for example code into a development environment. This search interface presents blocks of example code, augmented with *running examples* and *written descriptions* when available (see Figure 1). In a between-subjects comparison with 20 participants, we found that Blueprint enables participants to search for and select example code significantly faster than with a standard Web browser.

Second, this paper introduces a technique for retrieving relevant example code from the Web for a user's query. To maximize speed, breadth, and ranking quality, the Blueprint server leverages a general-purpose search engine. Example code, descriptions, and running examples are then automatically extracted using a series of heuristic classifiers. By caching the results of this extraction, we are able to respond to user's queries at interactive rates.

Brandt and colleagues found that programmers used the Web with a range of intentions [3]. On one end of the spectrum, programmers used the web for *just-in-time learning* of skills, such as a new language or programming paradigm. On the other end of the intention spectrum, programmers used the Web for highly directed *reminders*. In these cases, programmers knew exactly what code needed to be written but chose to search for and copy examples to avoid typing. In between these two extremes, Brandt et al. observed programmers using the web to *clarify* existing knowledge. For example, Web search served as a "translator" when programmers didn't know the name of a function. Other times, programmers knew that there were multiple approaches to a task, and used Web search to quickly enumerate possibilities.

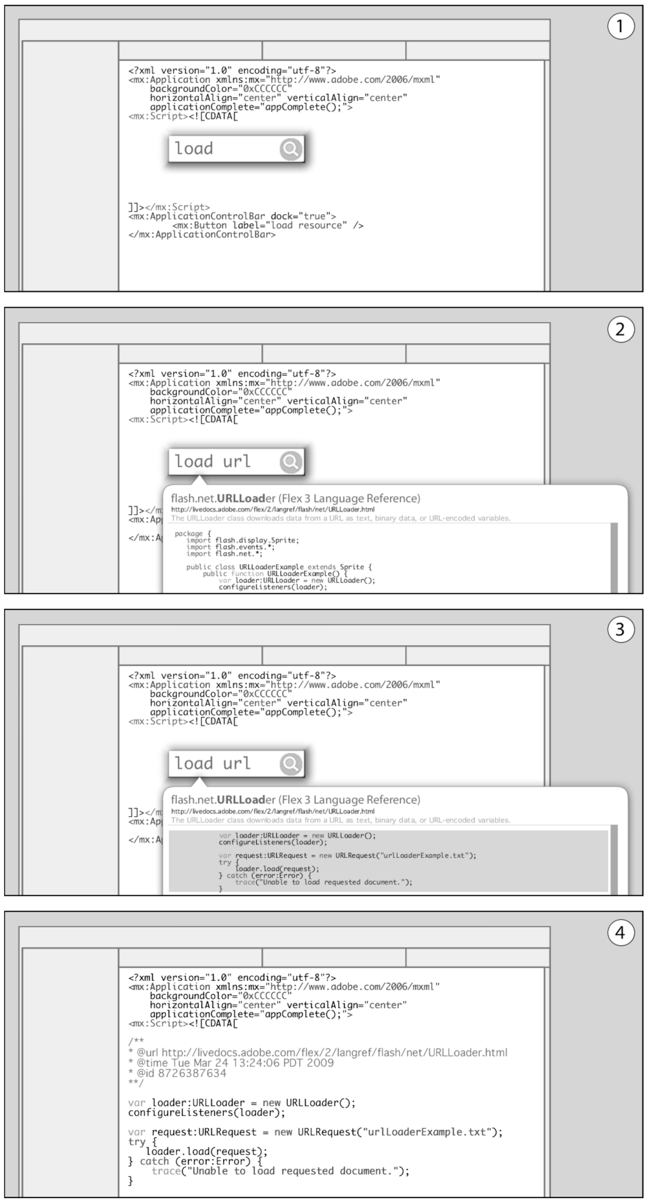
Blueprint is primarily designed to support *reminder* and *clarification* tasks. In these tasks, the source code is both the most useful representation when evaluating possible results and the information that the user desires.

The remainder of this paper proceeds as follows. First, to motivate Blueprint's interface choices, we offer background information on how programmers use the Web to inform our design. We then present a scenario enabled by Blueprint, and describe its implementation and evaluation. Next, we offer a discussion of the design space of tools to support programmer Web use, and position Blueprint within this space to better understand its strengths and limitations. We close with a survey of related work and thoughts on future research directions.

SCENARIO: DEVELOPING WITH BLUEPRINT

Jenny is a web programmer at a power utility that is about to launch a campaign encouraging individuals to lower their power consumption. She is prototyping a web application for customers to compare their daily power consumption to average levels. Her functional prototype should: *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 customer-specific and average power-usage data. The company already has a Web service that returns XML-formatted power usage data for a given customer. Jenny has written code to fetch data from the Web before, and is capable of constructing the necessary code. However, she thinks it will be faster to find and copy an example than to rewrite the code from scratch. She



```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    backgroundColor="0xCCCCCC"
    horizontalAlign="center" verticalAlign="center"
    applicationComplete="appComplete();">
<mx:Script><![CDATA[

]]></mx:Script>
<mx:ApplicationControlBar dock="true">
        <mx:Button label="load resource" />
</mx:ApplicationControlBar>
```

flash.net.**URLLoad**er (Flex 3 Language Reference)
http://livedocs.adobe.com/flex/2/langref/flash/net/URLLoader.html
The URLLoader class downloads data from a URL as text, binary data, or URL-encoded variables.

```
package {
    import flash.display.Sprite;
    import flash.events.*;
    import flash.net.*;

    public class URLLoaderExample extends Sprite {
        public function URLLoaderExample() {
            var loader:URLLoader = new URLLoader();
            configureListeners(loader);
```

```
            var loader:URLLoader = new URLLoader();
            configureListeners(loader);

            var request:URLRequest = new URLRequest("urlLoaderExample.txt");
            try {
                loader.load(request);
            } catch (error:Error) {
                trace("Unable to load requested document.");
            }
```

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    backgroundColor="0xCCCCCC"
    horizontalAlign="center" verticalAlign="center"
    applicationComplete="appComplete();">
<mx:Script><![CDATA[

/**
* @url http://livedocs.adobe.com/flex/2/langref/flash/net/URLLoader.html
* @time Tue Mar 24 13:24:06 PDT 2009
* @id 8726387634
**/

var loader:URLLoader = new URLLoader();
configureListeners(loader);

var request:URLRequest = new URLRequest("urlLoaderExample.txt");
try {
   loader.load(request);
} catch (error:Error) {
    trace("Unable to load requested document.");
}
```

Figure 2. Example-centric programming with Blueprint. To initiate blueprint, the user press 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 his project (3). Blueprint automatically adds a comment containing metadata that links the example to its source (4).

presses a hotkey to invoke the Blueprint search dialog; a search box appears next to her cursor (see Figure 2, step 1). This interface extends auto-completion techniques [12] to presents entire blocks of example code as results instead of variable and method names. She types *URLLoader*, the name of the main class associated with this task. Blueprint knows the language and framework version she is using, and returns appropriate examples (step 2) These results are presented below the search box. She flips through the first few examples (step 3) and sees one that creates an XML object out of the data that is returned. She presses *Enter*, and the code is pasted in her project (step 4). Along with the code, Blueprint adds a special machine- and human-readable comment that records the URL of the source and the date of copy. Every time Jenny opens this source file in the future, Blueprint will check this URL to see if the original example has changed (*e.g.,* if a bug is fixed), and will notify her when it does. She runs her code in Flex's debugger to confirm the XML has loaded and to inspect its format.

Next, she wants to change the user's mouse cursor to a busy cursor while the data is being loaded. She does not know the exact name of the relevant classes or methods involved, so autocomplete is no help. She places her cursor inside the function that initiates the data request and invokes Blueprint. This time, she searches for "busy cursor". She looks over the code in the first example returned, and sees the line "CursorManager.setBusyCursor()." She selects this line and presses *Enter* to paste it into her project. Now, knowing that *CursorManager* is the relevant class, she's able to use the standard autocomplete tool to write the line of code necessary to restore the cursor at the end of the data transfer.

Finally, Jenny wants to explore different charting components to display the data. 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. After browsing and refining her search, she settles on a line chart. She copies the example code from the Blueprint panel into her project and modifies it to bind the chart to the XML data. Her prototype is now complete.

IMPLEMENTATION

Blueprint's implementation comprises two parts: the *client-side plug-in*, which provides the user interface for searching and browsing results, and the *Blueprint server*, which executes the searches for example code.

Client-Side Plug-In

The Blueprint client is a plug-in for Adobe Flex Builder [13]. Flex Builder, in turn, is a plug-in for the Eclipse Development Environment [14]. The Blueprint client provides three main pieces of functionality. First, it provides a user interface for initiating searches and displaying results. Second, it augments users' queries with contextual information (*e.g.* programming language and framework version) before forwarding them to the Blueprint server. Third, it notifies the user when the Web origin of examples they adapted has updated (*e.g.,* when a bug is fixed).

User Interface

Much of Blueprint's interface is implemented using HTML that is rendered by SWT Browser widgets. Communication with the Blueprint server occurs over HTTP using the JSON data format [15]. Creating the interface with HTML facilitated rapid iteration; JSON's broad cross-language support facilitates implementing different components in different languages.

To facilitate learning, Blueprint employs the same syntax highlighting and navigational mechanisms as the development environment and existing autocomplete tools. Users can navigate through examples using the Tab key and copy/paste the selected example by pressing enter.

Augmenting Queries

The Blueprint client augments user queries with contextual information to increase the results' relevancy. The current prototype augments the query with the programming language name and version (here, "Flex 3"). In future work, it would be interesting to explore the benefits of adding additional contextual information (*e.g.* the types of local variables currently in scope.) Holmes and colleagues have explored this idea in the context of retrieving example code from a fixed repository [16]; it is not immediately clear how this would generalize to Web search.

When a user pastes example code into their project, Blueprint inserts a comment at its beginning – much like a Javadoc comment [17]. This comment tags the example code with the *URL* it was taken from, the *date and time* it was inserted, and a *unique numerical identifier*. This metadata is both human and machine-readable, so users can for example return to the URL. Also, embedding the metadata in the source file simplifies file management. Blueprint searches for these example comments each time a file is opened. For each comment, it queries the Blueprint server to find out if the original example has been modified since it was copied.

Blueprint Server

The Blueprint server responds to queries for example code. 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 search appliance. This appliance indexes Flex-specific content from across the Web. When Blueprint receives a query, it hands the query off to this search engine, which returns a set of URLs. The challenge of this approach is that standard search engines return URLs of *pages*, yet Blueprint's goal is to return *examples*. For each URL, Blueprint retrieves the corresponding page, parses it, and extracts source-code examples. In order for the Blueprint server to be responsive, processing and transforming the results from the underlying search engine cannot incur significant latency. To improve response time, the server caches page contents and parsing results.

The final step of a query is for the server to return a set of examples to the requesting client. Each example comprises unformatted and syntax-highlighted versions of the code, a description of the code, the URL that the code comes from, and, when possible, the URL of a running example of the code. Blueprint produces the syntax-highlighted version using Pygments [18], which outputs model-based markup. The client transforms this markup into styled code using CSS.

Extracting example code, descriptions, and running examples

To facilitate locating and extracting source code from Web pages, Blueprint first segments the page. Then, Blueprint classifies each segment as being source code or not. This offline processing takes the current Blueprint prototype about 10 seconds per page.

Since HTML documents often contain errors (*e.g.*, missing tags or extraneous quotes), Blueprint first transforms them into proper XHTML documents so that we can leverage their structure in the segmentation process. We preprocess the HTML file with the BeautifulSoup library [19], which generates valid XHTML output for any input.

Next, Blueprint 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. We also extract SCRIPT and OBJECT blocks as block-level elements, because running examples are usually contained within these tags. To find block-level elements, Blueprint traverses the document depth-first. When we reach a leaf element, we backtrack to the nearest containing block-level element and create a segment. If the root of the tree is reached before finding a block-level element, the element immediately before the root is extracted as a segment. This algorithm keeps segments ordered exactly as they were in the original document.

Third, Blueprint strips each segment of the majority of its formatting so that we can reliably determine whether or not it contains example code. Although formatting, such as SPAN elements for syntax highlighting, helps with the segmentation process, it complicates source-code recognition. For readability, Blueprint preserves the original line breaks. To strip formatting while retaining line breaks, we "render" each segment to plain text using w3m, a text-based web browser [20]. 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). This possible over-segmenting is not a problem as long as blocks of example code are being correctly parsed into single segments.

We now have clean, separate segments that are ready for classification. There are two main approaches for classifying a segment as code or not. The first is to parse each segment with the languages of interest. Segments that parse correctly are considered code. For Blueprint, this would be ActionScript and MXML, the two languages used by Adobe Flex. In practice, this approach yields many false negatives. For example, code with line numbers or a single typo will cause parsing to fail. The alternate approach is heuristic-based classifiers that look for a high occurrence of features unique to code, such as curly braces, frequent use of language keywords, and lines that end with semi-colons [11]. This approach has many fewer false negatives, but includes more false positives, such as text that discusses code. Blueprint uses a heuristic-based approach because, in this domain, false positives (spurious results) are strongly preferable to false negatives (missing results).

The next step is to extract descriptions and, where possible, running examples for each code segment. Informal inspection of pages containing example code revealed two patterns: 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).

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.

Caching and pre-populating the example database

Blueprint caches the extraction results so that examples can be returned immediately. Without caching, search response time would be bounded by the amount of time required to retrieve the resulting Web pages. URLs returned by the underlying search engine that are not in our cache are ignored for that query. They are then added to the cache by a background process so they can be used in future queries. A future version of Blueprint could return these to users asynchronously.

To make Blueprint usable initially, 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.

Keeping track of changes to examples

Each time a page is crawled, Blueprint checks for updates to the examples (*e.g.,* bug fixes). It compares old and new examples using the *diff* tool. Because pages typically contain fewer than ten examples, Blueprint compares all pairs of examples on the new page and examples on the old page. If the examples match 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. Storing examples with timestamps facilitates Blueprint's ability to track changes and to return all versions of an example so that a user can see what has changed.

EVALUATION

We conducted a laboratory study to better understand how Blueprint affects the example-centric development process. This 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 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.

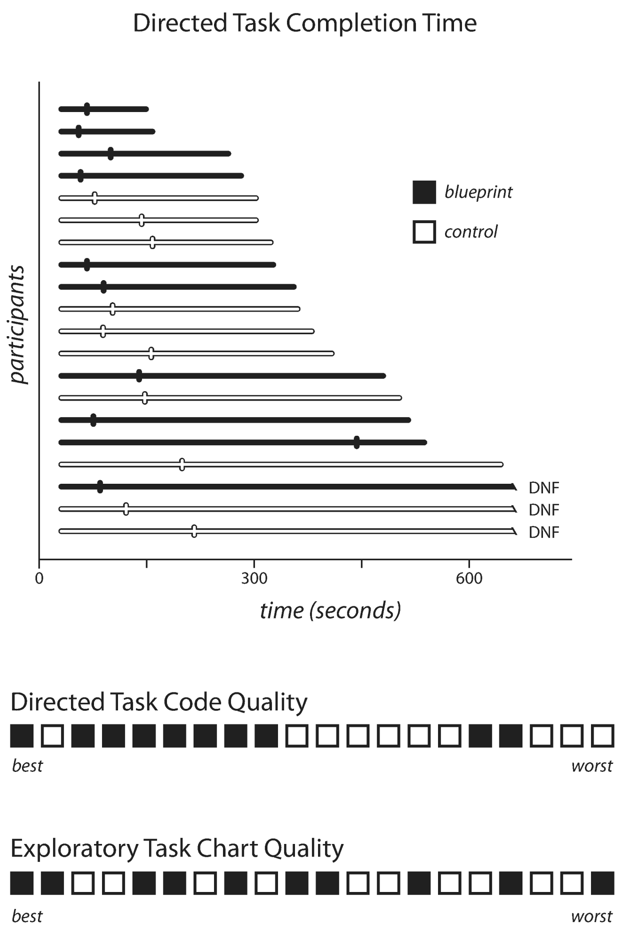


Figure 3. Results from the study. The top graph shows each participant's completion time on the directed task. Time of first paste is shown as a tick mark; participants who used Blueprint are shown in black. The two bottom graphs show ranking of participant's code quality (directed task) and chart quality (exploratory task), respectively. Again, participants who used Blueprint are shown in black.

Method

Twenty professional programmers (16 men, 4 women) 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.

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 possible best 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 3.

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* had poor scent. For this reason, we speculate that some control participants overlooked it in favor of a result with better scent.

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.

Suggestions 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.

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.

Finally, we noticed that some participants had difficulty locating specific lines they were looking for within examples. Two interface changes could improve this process: First, search terms should be highlighted within the results. Second, users should be able to search *within* the result set.

DESIGN SPACE

This section discusses the important decisions made in Blueprint's design, and positions them in a space of alternative designs (see Figure 4). Positioning Blueprint within this space helps structure a discussion of Blueprint's limitations and suggests fruitful areas for future work.

Task: Blueprint is expressly designed to facilitate implementing new functionality. Programming involves many tasks: planning and design, implementation, and testing and debugging. For many other tasks—*e.g.* deciphering a cryptic error message during debugging—it might make more sense to initiate searches from the program's output. In such situations, a code-centric view of results might hide important information.

Knowledge: Blueprint presents a code-centric view of results, so programmers must be knowledgeable about the tools (*i.e.* languages and frameworks used in the example) to interpret the results. For any given task, programmers have a certain amount of knowledge about both the tools

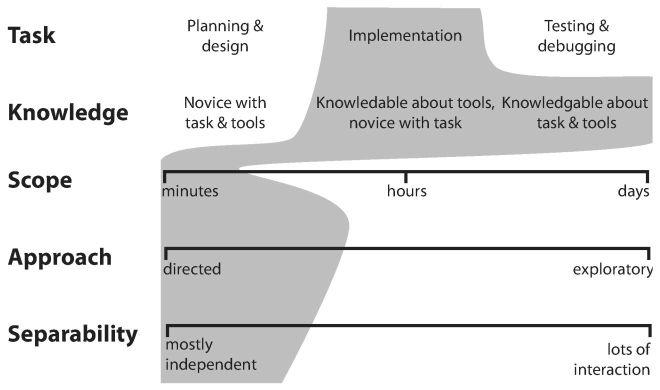


Figure 4. Design space of tools to aid programmers' Web use. Blueprint is designed to address the portion of the space shown with a shaded background.

they are using (*e.g.* languages and libraries), and the task itself (*e.g.* implementing a piece of functionality).

Scope: We designed Blueprint to make small tasks faster. Blueprint's primary interaction behaves similarly to auto-completion [12] to make inserting small blocks of code more efficient. The transient nature of this interface makes it impractical for larger tasks. Blueprint does support docking of results as a persistent panel. However, this panel lacks the rich navigational features present in modern Web browsers, (*e.g.* tabs, history, bookmarking, and finding text within a page). This may be useful in completing larger tasks.

Approach: Currently, Blueprint works best for directed tasks. There are two reasons for this: First, Blueprint requires users to search, rather than browse. Users must have a concrete notion of what they want to do in order to construct a query. Second, Blueprint currently contains little support for comparing results, a crucial part of exploratory tasks. At times, programmers know exactly what they want to do. At others, they are interested in exploring a wide range of possibilities. Providing better support for comparison and evaluation is an important direction for future work.

Separability: Because Blueprint inserts example code directly into the user's project, it provides the most benefit when example code requires little modification. As such, it is most useful for tasks that are largely independent from other pieces of the project. When a piece of code is deeply intertwined with the larger context it is embedded in, it may be easier to create the new instance while referring to the example rather than copying it.

The approach Blueprint introduces leads to broader questions about example-centric development.

Is enough example code available online?

It seems to depend on both language and task. The most popular web programming languages have rich online resources. This is likely due to the background of the community using these languages: it is natural for them to put code on the web.

Languages that provide a comprehensive standard library (*e.g.,* Python and Java) seem to have more code online than those that do not (*e.g.,* C). This is perhaps because it is easier to write small blocks of code that implement non-trivial functionality without requiring external libraries. The high cost of installing and linking external libraries may often outweigh the benefits associated with co-opting examples that use them.

Finally, common functionality is more likely to be available online than rare functionality.

Is the context of an example important?

Blueprint extracts example code from Web pages and presents it to the user outside of its original context. There are trade-offs associated with this decision. On the one hand, presenting all results using a consistent interface may speed up understanding [21]. On the other hand, programmers sometimes use a page's high-level visual features (*e.g.* whether a page contains advertisements) to make initial judgments about the quality of example code [3]. Programmers most commonly use surface features about the code's context when they have difficulty evaluating the code quality itself [3], *e.g.* when selecting a tutorial to learn about a new language or programming paradigm. This is consistent with general novice/expert distinctions [22]. Blueprint is designed primarily for situations where the programmer has enough knowledge about the tools he is using to evaluate the code itself.

RELATED WORK

This research is inspired by prior work on tailoring Web search interfaces, and providing support for example-centric development.

Tailoring Web Search for Specific Tasks

Prior work has explored how programmers search the Web [3, 11] and tailoring Web search to their needs [9-11, 23, 24]. When building Assieme, a search engine for Java programmers, Hoffmann and colleagues recognized the important role that sample code plays in helping a developer select between APIs [11]. Assieme's search result view allows the user to view example code for any API by hovering his mouse over individual results. Blueprint carries this idea further by making example code the primary feature of the result set. Blueprint also builds on Assieme's method of extracting example code from Web pages by using heuristic-based classifiers.

Blueprint's approach of 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.* [3, 11]). First, it is substantially more resource-efficient to implement. Keeping a document collection up to date is expensive, and this is handled automatically in our approach. Second, it is an understatement to say that generating high-quality search results from natural-language queries is a hard problem. It may be unreasonable to assume that, even with a restricted search domain, we can do a better job than commercial search engines.

Finally, using a general-purpose search engine to provide data has the direct consequence that most examples are taken from tutorials, blogs, and API pages. We believe this my be beneficial because these examples are more likely to be written in a way that is easy to understand than examples extracted from large source code repositories.

More broadly, there has been recent interest in providing alternative representations for Web search results [21, 25-29]. Dontcheva and colleagues' work on search templates introduced the idea of creating customized views of Web pages called "cards" [21]. By defining a set of relations between Web page elements and elements on the card, users can then view a diverse collection of Web search results using a consistent interface. Blueprint works similarly: common elements (code examples) from diverse Web pages are automatically extracted and presented in a consistent interface for faster browsing and selection.

Example-Centric Development

Prior work has created tools to assist with example-centric development [30]. This work has addressed the availability of example code problem by mining code repositories [31] or synthesizing example code from API specifications [32]. Blueprint is unique in that it uses regular Web pages (*e.g.* forums, blogs, and tutorials) as sources for example code.

Using tutorials and blogs 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 to find the code they are looking for.

In the majority of prior work, the programmer must already have some language- or API-specific "handle" to the code he wants to retrieve. An exception is the d.mix system; it enables users to "sample" a Web page's UI elements to yield the API calls necessary to create them. Future work could explore extending Blueprint so that searches for example code could be initiated by direct manipulation of another program's interface or output.

Little and Miller's research introduced techniques for programmers to write keyword code that is transformed into syntactically correct statements through a search process [33]. Future work could combine ideas from Blueprint and keyword programming.

CONCLUSION AND FUTURE WORK

We have presented a user interface for accessing online example code from *within the development environment*. This interface displays search results in an *example-centric manner* to support programming by example modification. This paper described the implementation of Blueprint, a lightweight method for using a general-purpose search engine to create code-specific search results that include written descriptions and running examples. Empirical results suggest that Blueprint's approach of integrating web search into the development environment helps programmers acquire and adapt online resources more efficiently.

An important avenue for future work is to improve the modification of example code. Copied code can introduce bugs when programmers assume that sample code works and forget to adapt portions of the example. Blueprint users would benefit from rich refactoring support for pasted code. This would help users change variable names consistently and reduce the number of errors. It might be valuable to rethink the character-at-a-time editing paradigm entirely. Would it be more efficient to navigate pasted code a token at a time? Perhaps arrow keys should move the user's cursor between tokens, and typing over top of an existing token should automatically replace all occurrences of that token within the pasted region.

While example-centric development is common, there is little aggregated knowledge about how users adapt examples. If Blueprint could show users how code has been changed in the past, perhaps they'll make fewer errors. For example, if all ten previous users changed a literal, it is highly likely that the eleventh user should change this literal as well. The wisdom of the crowds may enable significant advances in online programming tools.

REFERENCES

1. Brooks, F.P., *The Mythical Man-Month: Essays on Software Engineering*. 1995: Addison-Wesley.
2. Pirolli, P.L.T., *Information Foraging Theory*. 2007, Oxford, England: Oxford University Press.
3. Brandt, J., et al. Two Studies of Opportunistic Programming: Interleaving Web Foraging, Learning, and Writing Code. In *Proceedings of CHI: ACM Conference on Human Factors in Computing Systems*, Boston, Massachusetts, 2009.
4. Brandt, J., et al. Opportunistic Programming: How Rapid Ideation and Prototyping Occur in Practice. In *Proceedings of WEUSE: International Workshop on End-User Software Engineering*, p. 1-5, Leipzig, Germany, 2008.
5. Hartmann, B., S. Doorley, and S.R. Klemmer. Hacking, Mashing, Gluing: Understanding Opportunistic Design. 2008.
6. Clarke, S. What is an End-User Software Engineer? In *End-User Software Engineering Dagstuhl Seminar*, Dagstuhl, Germany, 2007.
7. Nardi, B.A., *A Small Matter of Programming: Perspectives on End User Computing*. 1993: The MIT Press.
8. deHaan, P. *Flex Examples*. http://blog.flexexamples.com/
9. *Google Code Search*. http://code.google.com
10. Stylos, J. and B.A. Myers. Mica: A Web-Search Tool for Finding API Components and Examples. In *Proceedings of VL/HCC 2006: IEEE Symposium on Visual Languages and Human-Centric Computing*, p. 195-202, 2006.

11. 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*, p. 13-22, Newport, Rhode Island, 2007.
12. *Microsoft IntelliSense*. http://www.microsoft.com/visualstudio/
13. *Adobe Flex*. http://www.adobe.com/flex
14. *Eclipse*. http://www.eclipse.org
15. *JSON Data-Interchange Format*. http://json.org
16. Holmes, R. and G.C. Murphy. Using Structural Context to Recommend Source Code Examples. In *Proceedings of ICSE: International Conference on Software Engineering*, p. 117-125, 2005.
17. *Javadoc*. http://java.sun.com/j2se/javadoc/
18. *Pygments*. http://pygments.org/
19. Richardson, L. *Beautiful Soup*. http://www.crummy.com/software/BeautifulSoup
20. *w3m*. http://w3m.sourceforge.net
21. Dontcheva, M., et al. Relations, Cards, and Search Templates: User-Guided Web Data Integration and Layout. In *Proceedings of UIST: ACM Symposium on User Interface Software and Technology*, p. 61-70, Newport, Rhode Island, 2007.
22. Chi, M.T.H., P.J. Feltovich, and R. Glaser, *Categorization and Representation of Physics Problems by Experts and Novices.* Cognitive Science, 1981. **5**(2): p. 121-152.
23. Bajracharya, S., 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*, p. 681-682, Portland, Oregon, USA, 2006.
24. *Krugle*. http://www.krugle.com
25. Woodruff, A., et al. Using Thumbnails to Search the Web. In *Proceeding of CHI: ACM Conference on Human Factors in Computing Systems*, p. 198-205, Seattle, Washington, 2001.
26. Dontcheva, M., et al. Summarizing Personal Web Browsing Sessions. In *Proceedings of UIST: ACM Symposium on User Interface Software and Technology*, p. 115-124, Montreux, Switzerland, 2006.
27. Medynskiy, Y., M. Dontcheva, and S.M. Drucker. Exploring Websites through Contextual Facets. In *Proceeding of CHI: ACM Conference on Human Factors in Computing Systems*, Boston, Massachusetts, 2009.
28. Teevan, J., et al. Visual Snippets: Summarizing Web Pages for Search and Revisitation. In *Proceeding of CHI: ACM Conference on Human Factors in Computing Systems*, Boston, Massachusetts, 2009.
29. Adar, E., et al. Zoetrope: interacting with the ephemeral web. In *Proceedings of UIST: ACM Symposium on User Interface Software and Technology*, p. 239-248, Monterey, California, 2008.
30. Hartmann, B., et al. Programming by a Sample: Rapidly Creating Web Applications with d.mix. In *Proceedings of UIST: ACM Symposium on User Interface Software and Technology*, p. 241-250, Newport, Rhode Island, 2007.
31. 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*, p. 413-430, 2006.
32. Mandelin, D., et al. Jungloid Mining: Helping to Navigate the API Jungle. In *Proceedings of PLDI: ACM SIGPLAN Conference on Programming Language Design and Implementation*, p. 48-61, Chicago, IL, USA, 2005.
33. Little, G. and R.C. Miller. Translating Keyword Commands into Executable Code. In *Proceedings of UIST: ACM Symposium on User Interface Software and Technology*, p. 135-144, Montreux, Switzerland, 2006.