PAPER 9

Abstract—**The web is an important source of development-related resources, such as code examples, tutorials, and API documentation. Yet existing development environments are largely disconnected from these resources. In this work, we explore how to provide useful web page recommendations to developers by focusing on the problem of refinding web pages that a developer has previously used. We present the results of a study about developer browsing activity in which we found that 13.7% of developers visits to code-related pages are revisits and that only a small fraction (7.4%) of these were initiated through a low-cost mechanism, such as a bookmark. To assist with code-related revisits, we introduce Reverb, a tool which recommends previously visited web pages that pertain to the code visible in the developer's editor. Through a field study, we found that, on average, Reverb can recommend a useful web page in 51% of revisitation cases.**

I. Introduction

Software developers use information resources on the web to help perform many different programming tasks, including learning new programming concepts, reminding themselves of syntactic programming language details and clarifying error messages, amongst others [1]. Despite the prominence of web search and exploration in the workday of many software developers, the tools of software development and web browsing remain disconnected from each other. In practice, retrieval of relevant web pages is an effortful process, requiring a developer to choose a set of search keywords, enter the keywords in their browser, evaluate multiple pages returned from a search and rinse and repeat until appropriate resource(s) are found.

Ideally, the cost of finding appropriate web pages for a developer's task would be negligible. If appropriate web pages could simply be fetched and be readily available based on the developer's current work, a developer could remain focused on their task-at-hand. The Fishtail tool takes this approach: Fishtail extracts keywords from a developer's task context (all programming constructs on which the developer worked as part of a task [2]), automatically queries the web with those keywords and then recommends a selection of pages from those returned from the search [3]. While this approach greatly reduces the cost of searching for relevant web pages, Fishtail is unable to recommend appropriate pages with high accuracy. As a result, the cost of perusing recommended pages for those appropriate for the current task remains high, despite some automation in the creation and execution of a search.

One way to potentially improve the relevancy of web pages recommended is to limit page recommendations to those that a developer has previously found useful. In essence, predict likely revisitations of web pages. Although such an approach would not be as ideal as recommending appropriate web pages from the wild, it may still provide value if developers commonly revisit pages; studies on general web page revisitation (i.e., including commonly visited pages such as email and calendar) have found revisitation rates of between 45% [4] and 81% [5]. Unfortunately, no studies have looked at revisitation for code-related pages. To investigate code-related revisitation rates, we report in this paper on a study of eleven software developers in which we analyzed the developers' web browsing history for a three-month period. Ignoring revisits that occurred shorly after an initial visit (i.e., within 15 minutes), we found a mean code-related revisitation rate of 13.7%. Interestingly, 45% of participants had at least three one-hour periods containing three or more code-related revisits and 18% had nine or more such periods. As each of these revisits can require substantial work in re-finding the web pages of interest from web searches, we believe there is value in a tool that can predict revisits of code-related pages.

In this paper, we introduce such a tool, called Reverb, that pro-actively recommends web pages previously used by a developer as the developer works on source code in a development environment. Reverb detects code-related web pages perused by the developer in a web browser, indexes the keywords used on those pages, and then recalls and displays pages that are similar to code under active development by the developer. The recommendations made by Reverb are ranked and presented based on similarity in the content of the code being worked on and the page, and the frequency and recency (frecency) of access to the page by the developer.

To determine if Reverb can provide benefit to software developers, we conducted a field study during which nine participants used Reverb in their own development tasks for a period of six Java programming hours. We found that Reverb, on average, was able to predict a code-related web page revisted by the developer in 51% of the cases of revisitation. For 33% of the developers in our study, the prediction rate was over 67%. We believe the performance of Reverb shows the promise of using local code context to drive the pro-active recommendation of code-related web pages.

This paper makes three contributions:

- it provides empirical data about how often software developers return to code-related web pages and the methods they use to perform these revisits,

- it introduces a tool, Reverb, to support developers in refinding code-related web pages, by proactively recommending pages from the developer's own browsing history that relate to the code under development, and
- it presents the results of a field study of Reverb that suggest that local code context is helpful in recommending useful web pages.

We begin with a review of related work on general web page revisitation and specific support for software developers using the web (Section II). We then describe the results of a study into how often code-related web page revisits happen for software developers (Section III) and introduce the Reverb tool that we have developed to support the proactive recommendations of code-related pages (Section IV). We follow the tool description with the results of a field study of the tool (Section V), before concluding with a discussion of our approach (Section VI) and a summary of the paper (Section VII).

II. Related Work

A number of studies have considered general web page revisitation. Tauscher and Greenberg tracked six weeks of web usage of 23 participants to compute the likelihood—called the recurrence rate—that a given page visit was a revisit [6]. In their study, they observed a recurrence rate of 58%. McKenzie and Cockburn analyzed the history of 17 web users, finding a recurrence rate of 81% [5]. This higher recurrence rate may have been due to a data preprocessing step in which URLs were truncated, thus ignoring query parameters and increasing the number of pages to be considered a revisit. Obendorf and colleagues conducted a long-term study of 25 browser users [4]. They reported a recurrence rate of 45.6% based on distinct URLs, which included query parameters and POST data. Adar and colleagues gathered anonymized browsing information from 612,000 users of the Windows Live Toolbar over a five-week period. [7]. Instead of computing the recurrence rate, Adar and colleagues focused on identifying revisitation patterns associated with particular web pages. The study of code-related revisitation on which we report in this paper (Section III) is most similar in style to Obendorf and colleagues. In focusing on revisitation of one kind of page, code-related pages, our study shares similarities with the work of Adar and colleagues. Whereas our focus is on recommending pages to revisit based on a developer's current work, Adar and colleagues focused on understanding user behaviour from how pages are revisited.

Prompted by the recurrence rates found in the empirical studies described above, researchers have devised a number of techniques to assist users in refinding web pages. Most of these approaches enrich the existing functionality of the browser, such as extending the functionality of the back button [8]. Others have considered the recurrence of specific navigation trails in users' browsing histories (e.g., [9]). For example, a visit to a user's web-based email client may frequently be followed by a visit to an online calendar. A recent study found that augmenting a frequency- and recency-based ranking algorithm with page transition probabilities computed from the user's browsing history increased the percentage of successful revisit predictions from 71.7% to 81.8% [10]. Reverb also uses contextual information to predict pages that a user may wish to revisit; instead of using the context of previous navigation, Reverb relates the content of the developer's code editor with similar web pages in the user's browsing history.

A variety of tools have been proposed to assist developers in their use of the web. Mica [11] and Assieme [12] facilitate searching for code-related resources on the web. These tools allow a developer to enter a task-oriented query (e.g., "java create zipfile") and then categorize the search results by the APIs and types that they reference. Reverb differs in intent from these tools in eliminating the need for a search to be performed in at least some situations. The Codetrail tool watches the pages a developer browsers to, automatically detects if a page has API documentation and links that documentation to the source code for easier access [13]. In addition, Codetrail can detect blocks of code on visited pages that match code in the developer's workspace and can automatically generate a bookmark in the workspace to capture the link. Similar to Codetrail, Reverb surfaces web pages related to the developer's current work. In contrast to Codetrail, Reverb can surface useful web pages in more situations because it bases recommendations on automatically-generated queries rather than on temporal access and because it is not restricted to pages detected as describing an API. HyperSource maps web pages visited to corresponding source code edits and maintains these associations over time [14]. Consequently, HyperSource provides a precise picture of code provenance which is not available with Reverb. In contrast, Reverb's associations are based on shared keywords, rather than temporal locality, allowing a web page last visited in an earlier project to be recommended based on newly-added code. Finally, Blueprint embeds the web search interface in the development environment and integrates results from search atop the source code editor [15]. It does not provide results proactively like Reverb, but does use local context to augment the user-entered query, such as adding the name of the programming language.

III. How Many Visits are Revisits?

Do software developers revisit code-related web pages? If revisits occur, what is the recurrence rate? What mechanisms do developers use to revisit pages? These are the questions we needed to answer to understand if a tool that recommends previously visited web pages might have value for software developers. For instance, if revisits do not occur or are infrequent, little value might be gained from tool support. If revisits occur, but are through developer-created bookmarks, the cost of refinding the web pages would be minimal and little value might be gained from a tool. To answer these questions, we conducted a study of software developer browsing behaviour.

A. Method

We investigated the questions of interest by analyzing detailed browsing history databases maintained by the Mozilla Firefox[1] and Google Chrome[2] web browsers. Both browsers use the SQLite database engine[3] for managing browsing history. Each time the user visits a page, an entry containing the timestamp and a location ID is added to a table of page visits. The full URL of each page visit (including query parameters and anchor, if present) is stored in a separate table of locations. By analyzing browsing history, we only needed to have participants access data that was already collected, avoiding the risk of users changing their behaviour because they knew they were participating in a study.

We provided each participant in our study a tool that we had written which prompted the participant to choose a browsing history window of at least two months in which the participant was actively coding. The tool extracted a list of web page visits within that window from the browser's history database. Redirects and Google search page visits were filtered from the list. The tool then attempted to download the content of each page in the list. Downloaded pages were filtered to remove `<script>` elements as these elements are not displayed to the user and can contain tokens which could make a page appear code-related. The text content of the page was then extracted and the page classified as code-related or non-code-related using the heuristics described below. To estimate the accuracy of this classification, a random sample of 25 pages classified as code-related was presented to the participant and the participant was asked to flag any incorrectly-classified pages. A report was then uploaded to our server that contained the participant's anonymized browsing history, the page classifications and the manual classifications performed by the participant. The source code for the tool used by participants in this study is available on Google Project Hosting.[4]

Several possibilities exist for determining if a web page is code-related. One could try to identify code based on the density of delimiter tokens or presence of keywords from common programming languages. This approach would need to be customized for each language to be supported. Alternatively, an approach based on latent semantic indexing (LSI) could be used. However, this approach requires the training of the indexer on a corpus of documents. Furthermore, Bacchelli and colleagues found that LSI performed well on recall, but fared poorly in precision of its results [16]. We required an approach with high precision since we were using the classifier to determine the recurrence rate of code-related pages.

To achieve high precision, we chose an approach based on the camelCase naming conventions prevalent in current coding practice. We constructed a regular expression to match words containing a medial capital letter (capitals that are not the first letter in the word) or underscores; we refer to this as an identifier filter. Although the identifier filter was constructed to avoid matching ordinary capitalized words and acronyms, the filter generated many false positives due to company and product names on web pages. To avoid these false positives, we extended the filter to match only patterns that resembled method declarations or invocations with parameters. This pattern constrained of the identifier filter, followed by an open bracket, at most 40 non-close-bracket characters and a second instance of the identifier filter.

Restricting the filter to method declarations and invocations came at a further cost in recall. To compensate, we combined this with a pattern that looked for method declarations and invocations with no parameters. Experiments with our own browsing histories indicated that empty brackets were uncommon in non-code related pages. As a result, this pattern could allow identifiers without camelCase or underscores.

At least two matches to the two patterns—method declaration/invocation with parameters and without—had to be found for a web page to be classified as code-related. The full set of heuristics used in our classifier is available elsewhere [17].

[1]http://www.mozilla.org/en-US/firefox/new/, verified 16/08/12

[2]www.google.com/intl/en/chrome/browser/, verified 16/08/12

[3]sqlite.org, verified 16/08/12

[4]code.google.com/p/reverb-bookmarks, verified 16/08/12

B. Participants and Data

We recruited participants for this study from within the UBC Computer Science Department and from one software development company. Recruiting participants for this kind of study can be difficult given the reluctance of individuals to have their browsing history analyzed despite the fact that our study method uploaded only anonymized URLs. Of the eleven participants in this study, eight (73%) were graduate students and three (27%) were professional software developers. Seven participants (64%) used the Google Chrome web browser and four (36%) used Mozilla Firefox. Although Java was the programming language used by the largest number of participants (seven), a range of other programming languages were represented: C, C++, Javascript, Perl, Ruby, PHP and Python.

The average browsing history window used from these participants was 82 days. In total, we analyzed 906 days of browsing history.

C. Results

To answer the question of whether developers revisit code-related web pages, we compute the recurrence rate as defined in previous work [6]:

$$R = \frac{total\ URLs\ visited - different\ URLs\ visited}{total\ URLs\ visited} \times 100\%$$

This computation depends heavily on the definition of a URL. In our study, a page was identified by a full URL, including query parameters and fragment. For a visit to be classified as a revisit, the full URL had to be matched. However, if the URL was reached through a form submission, the POST parameters were not considered part of the page identifier. In addition, because the Firefox history database does not record visits initiated through the browser's forward and back buttons, these visits are not included in our calculations.

Over the 906 days of browsing history analyzed, our participants visited a mean of 6048 web pages of which a mean of 250 visits were code-related. The overall recurrence rate averaged across all participants was 41% (±14%), in-line with the findings of earlier studies [4]. For just those pages flagged as code-related, the recurrence rate was 23% (±12%). When code revisits occur close in time to the original visit, we think it is unlikely that tool support to help predict the revisit will be useful. More likely, a user will revisit the page based on a search results page that is still open in the browser, via a back or forward button or some similar mechanism. As a result, we also analyzed the browsing history data to understand recurrence rates for revisits occurring more than 15 minutes after a previous visit. With this 15-minute window, the overall recurrence rate is 27.3% (±11.0%) and the code-related recurrence rate is 13.7% (±10.6%).

We were also interested in gaining insight into the frequency of code-related page revisits. Using the 15-minute window, we measured the number of one-hour periods in which at least three code-related revisits were observed. For five (45%) participants, the count of these periods was three or greater. For two (18%) participants, the count was nine or greater. Because our classifier was optimized for precision over recall, we can consider that the actual code-related page revisits may be even higher.

Overall then, a reasonable number, greater than 13%, of code-related page visits are revisits. These revisits happen for some participants at a rate of more than three per hour. Given that revisits are occurring, we wanted to know which mechanisms developers were using to perform the revisits. To determine this information, we analyzed the browsing histories of the participants for code-related revisits where more than 15-minutes separated the visits to the pages. Because the Chrome history database provides a more fine-grained categorization of page transition types, we were only able to analyze the seven participants using Chrome. Figure 1 shows the breakdown of mechanisms used for revisits by these participants. Notable in this chart is the small percentage of revisits (3.7%) performed using bookmarks. The bookmark category represents visits initiated through a bookmark in the browser, including frequently-visited locations shown on Chrome's "New Tab" page. This small value shows that developers are not taking specific actions to remember code-related web pages. More often, in 27.7% of the cases, revisits fell into the Typed category, which captures visits initiated by typing into the address bar. Auto-complete is thus an important tool for developer to refind code-related web pages. The dominant mechanism at 54.4% for revisiting code-related pages was through a link from another page. Unfortunately, the anonymized history data we collected does not allow us to characterize the previous pages leading to these revisits. In some cases, these will be search results pages (instances of re-searching in the terminology of Obendorf et al. [4]). In other cases, these will be pages in a navigation trail the user follows

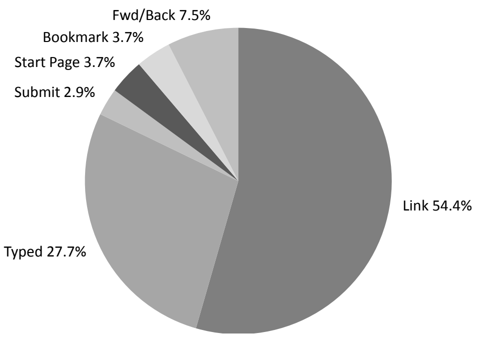


Fig. 1. Developer actions leading to code-related revisits

to refind the page (re-tracing). In either case, the developer is likely expending effort to peruse a search results page or to follow multiple links.

D. Limitations

The method we used to investigate code-related revisits may miss revisits to code-related pages. By having participants identify which of 25 randomly sampled pages marked as code-related were indeed code-related, we aimed to ensure that we did not over-estimate the code-related recurrence rate by including false positives. We chose this approach to minimize the effort our participants needed to expend on the study. However, it may have resulted in pages that were code-related being marked as not (i.e., false negatives). Our method also relies on the database maintained by the browser to determine which pages are visited. In some cases, we may not know that a developer re-consulted a page. For instance, if a page is in a browser tab and the tab is reactivated, we do not see this reactivation in the browser's database. We consider such cases low-cost revists and missing this information does not unduly affect our results.

Our participants represent a range of software development experience and settings, but are not representative of the total software developer population. The external validity of our results may thus be limited and the study should be extended to include a wider range of software developers. In particular, gaining a wider range of participants using a browser for which we can track how revisits occur, such as Google Chrome, would be beneficial to gain more insight into the costs associated with revisits.

Despite these limitations, this study provides the first insights into the behaviour of code-related revisits undertaken by software developers.

IV. Reverb: The Tool

Finding a web page needed for particular software development situation, such as a web tutorial about an API to understand how the API should be used or a project wiki page describing a feature being worked on, is a chore that takes a developer away from their task-at-hand. A developer typically has to use a search engine to find the appropriate page, but most search engines return a lot of results, requiring the developer to scan through potentially multiple pages of results to find the one or two most relevant web pages for their task. Doing this once to initially find a page is one thing, but having to do it over and over again to refind the page as you go back to work on similar code is tedious.

Reverb makes it easy to re-find these pages the developer has previously determined to be useful. Figure 2 shows Reverb in action. The tab labelled `WebPageDownloader.java` shows the code on which the developer is currently working. Reverb automatically detects and extracts code elements from the active viewport of the editor to form a query against the user's previous browsing history. In Figure 2, the code elements used for querying by Reverb are highlighted for clarity but Reverb does not typically highlight these elements. Reverb uses these code elements to query the user's browsing history. Pages whose content has a high similarity to these code elements and that might have been visited by the developer when working with on this or other similar code, are then ranked according to the frequency and recency with which a developer has visited the pages. The tab labelled `Reverb` in Figure 2 shows the display of results: the results are grouped under the query which the page matched. When an interaction with the code leads to the display of different code in the editor, Reverb refreshes the recommendations.

To function, Reverb must index the web pages visited by a developer, monitor activity in the code editor, form queries against the developer's browsing history and rank and group results for presentation to the developer. We describe each step in turn.

A. Indexing Web Pages

Reverb includes extensions for the Google Chrome and Mozilla Firefox browsers. These extensions monitor page loads and transfer the content of each page to Reverb's indexing service, which runs locally on the developer's computer. Only pages that are displayed for at least five seconds are sent to the indexer to ensure the page is likely of interest to the developer, to ensure that any dynamic content is loaded and to ensure that the browser extension does not impact page load time. We chose to index all web pages visited by the developer to ensure all pages with code would be included in the index. Based on data we have analyzed, the size of the index is unlikely to exceed 60MB in size with six months of browsing history.

Reverb's indexing service uses a general-purpose text indexing engine, Lucene[5], and is thus code-structure unaware. We chose this approach over a program structure aware indexing approach as used in search engines like Sourcerer [18] because structural links in web pages are often not resolvable. The Lucene records created by the service contain two fields: page title and page content. Preprocessing of the page prior to indexing is minimal: `<script>` elements are removed and then the text content of the remaining HTML elements is concatenated and indexed. The indexing service also maintains a database of locations visited. In addition to the last visit time and the number of visits, this database also contains a frecency column, which assigns a score to the web page based on frequency and recency of page visits (Section IV-C). An advantage of using Lucene is that it supports near real-time retrieval of newly-added content, meaning a page may be recommended in the development environment almost immediately after a developer has visited it for the first time.

[5]lucene.apache.org/core/, verified 16/08/12

B. Monitoring Activity and Forming Queries

Reverb also involves an Eclipse plugin that monitors mouse events in the Java code editor. Whenever the developer clicks in the editor, the plugin checks to see if the viewport has changed. If it has, the plugin retrieves the AST for the portion of the source code file that is visible and extracts the names of specific code elements. Reverb currently extracts the following elements:

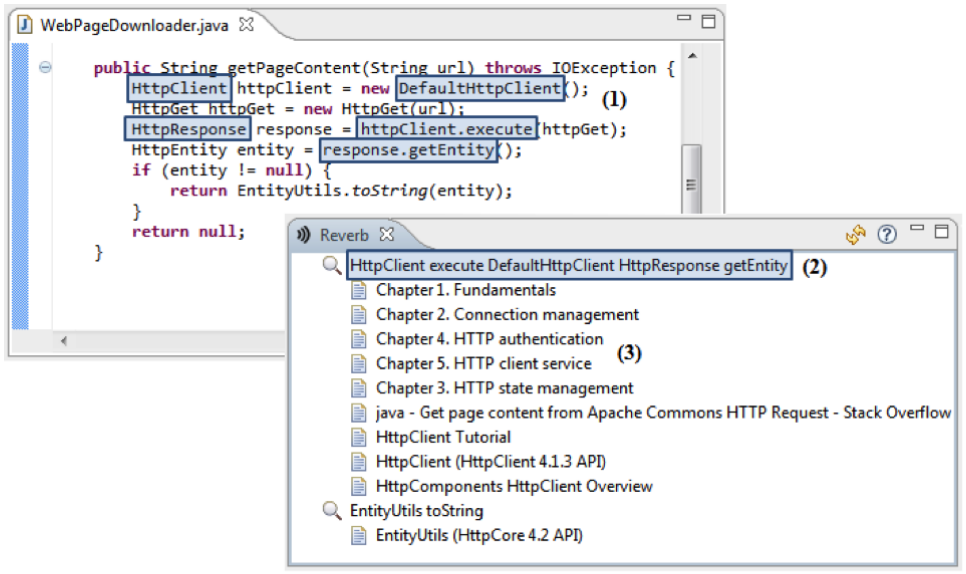
- type declarations,
- type references (e.g., in field, variable or parameter declarations),
- static field references,
- method declarations that override a method in a parent class or implement a method from an interface, and
- method invocations.

Once extracted, Reverb translates the code elements into a programming language-independent representation and sends them to the indexing service. In this representation, each code element is described by a structure containing four fields:

1) code element category (type declaration, type reference, static field reference, etc.),
2) package name (if code was compilable and information was available in the AST),
3) type name, and
4) method or field name (if applicable for the code element category).

Reverb does not distinguish between code elements that are part of the project's internal code base and those that belong to external libraries. Because many of the applications developers use to manage their workflow are now web-based, generating queries for internal code elements has an interesting side benefit: Reverb may suggest links to workflow items that relate to the current code, such as bug reports, code reviews, and internal wiki pages.

When Reverb's indexing service receives code elements, it constructs a Lucene query. Lucene's query processing uses a vector space model to match queries with documents (i.e., indexed web pages), but allows Boolean operators in the query to require specific terms to be present or absent. When the package name of a code elements is available, the query for a code element specifies that the document must either contain the fully-qualified name of the type (e.g., `org.apache.http.client.HttpClient`) or both the package name and the type name (e.g., `org.apache.http.client` and `HttpClient`). For



```
public String getPageContent(String url) throws IOException {
    HttpClient httpClient = new DefaultHttpClient();
    HttpGet httpGet = new HttpGet(url);
    HttpResponse response = httpClient.execute(httpGet);
    HttpEntity entity = response.getEntity();
    if (entity != null) {
        return EntityUtils.toString(entity);
    }
    return null;
}
```

```
HttpClient execute DefaultHttpClient HttpResponse getEntity
    Chapter 1. Fundamentals
    Chapter 2. Connection management
    Chapter 4. HTTP authentication
    Chapter 5. HTTP client service
    Chapter 3. HTTP state management
    java - Get page content from Apache Commons HTTP Request - Stack Overflow
    HttpClient Tutorial
    HttpClient (HttpClient 4.1.3 API)
    HttpComponents HttpClient Overview
EntityUtils toString
    EntityUtils (HttpCore 4.2 API)
```

Fig. 2. Reverb. The code elements highlighted in editor on the top left are used to build a query against the developer's browsing history. Result links shown in the bottom right are grouped according to the query they matched.

example, the Lucene query generated for calls to the `execute()` and `getParams()` methods of `HttpClient` is as follows:

```
+(org.apache.http.client.HttpClient OR
  (org.apache.http.client AND HttpClient)
     execute getParams
```

When the package name is not provided by the Eclipse plugin (because it is not available in the AST), the indexing service may still generate a query provided the type name is deemed selective on its own, using the identifier filter describe earlier (Section III). Multiple code elements associated with the same Java type (e.g., invocations of different methods belonging to the type) are grouped into a single query. Multiple queries may be generated if code elements extracted by the Eclipse plugin belong to multiple Java types.

C. Ranking and Grouping Query Results

The queries executed using Lucene result in zero or more web pages identified as relevant from the developer's browsing history. At a high-level, the pages resulting from the query are ranked according to the similarity of their content to the developer's current code, as well as the frequency and recency of page visits; the top ten ranked results are then grouped according to the types in the code that they match. We explain this process in more detail.

Given a query, Lucene assigns a relevance score to documents in its index according to how frequently the query terms appear in each document (i.e., Lucene uses a vector space model with TF/IDF nomalization). We made a few customizations to the default Lucene scoring function based on Reverb's specific requirements:

1) The Lucene scoring function allows for per-field boosts. Reverb assigns a multiplier of 3.0 to matches that occur in the title field of the page, as compared with matches in the page content, which have a boost of 1.0. We made this choice as pages with code element names in their titles are very likely to be code-related and relevant to the developer's current code.
2) The *coord* part of Lucene's scoring function boosts a result based on the fraction of the query terms that it matches. Result scores then vary approximately as the square of the number of query terms matched. Such a strong dependence is more appropriate for manually-generated queries, where the user expects results that match all of the query terms to be strongly favored. We forced the *coord* value to one, resulting in a dependence that was closer to linear.
3) The *lengthNorm* part of Lucene's scoring function boosts matches that occur in shorter documents over those that occur in longer documents. By default, the overall score is inversely proportional to the square root of the number of terms in the document. In initial experiments with our own browsing histories, this adjustment was unhelpful, dramatically favoring shorter web pages over longer ones. As with the *coord* value, we forced the *lengthNorm* to one.

Reverb gathers the top 20 results for each query run against the Lucene index and then processes all of those results to determine the top ten results which will be displayed to the developer. We perform this reduction in search results to ensure that we do not overwhelm the developer with recommendations. First, Reverb merges results from separate queries which correspond to the same web page. When merging, Reverb assigns a score to the page which is the sum of all Lucene relevance scores from the separate queries. This approach ensures that web pages that match more elements in the current code context score higher than pages that match fewer.

Next, Reverb adjusts result scores according to the frequency and recency of page visits. In general, frequency and recency of page visits are the variables most commonly used in predicting revisits. For example, in Firefox, the dropdown menu to the right of the address bar ranks pages according to a frecency score. A simple approach to deriving a frecency score is to sum over past visits to the page, weighting each one with an exponential time decay [19]. We follow this approach in Reverb. At time t, the frecency score for a web page is calculated as a sum over previous visit times t_i:

$$f(t) = \sum_i e^{-\tau(t-t_i)}$$

The time constant τ was chosen to ensure a fairly long half-life of six months, weighting frequency more heavily than recency. For scoring purposes, two visits six months ago are worth one today. We chose this value to ensure that if the developer started using a library again, a few months after the previous use, the tool would still be able to recommend pages visited during the earlier work. Tuning the value of this time constant is best done through empirical data; we discuss this tuning later in the paper (Section V).

To combine the Lucene relevance score and the frecency value of a page, we chose a geometric weighting scheme. The overall score v is calculated as the weighted product of the Lucene relevance r and the frecency f: $v = r^{\alpha} f$.

An advantage of a geometric weighting scheme over a linear one is robustness to inflation or deflation in the baseline values for either relevance or frecency. For example, with a linear scheme, if a new version of Lucene were released which tended to give higher relevance scores, the influence of frecency would be attenuated. We hope to eventually derive appropriate values for the weighting constant α from empirical data. For the initial version of the tool, we set $\alpha = 1$. However, we also set a hard maximum on f of 5. We were concerned about the potential for frequently visited pages to outscore more relevant ones, and tried to control this risk with a hard cap on frecency.

Finally, we group the top ten results together that are highly ranked according to the queries they matched as follows:

1) since a given result may have matched more than one query, each result is first associated with the query that gave it the highest Lucene relevance score,
2) then queries are ranked according to their highest scoring result page, and finally
3) if all of the hits for a given query also match a higher-ranked query, then the results for the lower-ranked query are merged with those of the higher-ranked query.

The merging step allows results for different, but related types to be merged under specific conditions. In practice, we have found that this algorithm leads to reasonable groupings. Types that appear in the same web pages, because they are used together frequently, will tend to be grouped together in the result list. This grouping strategy is easy to implement, and seems likely to give more meaningful groupings than one that relies on, for example, package structure. In the Reverb view, the header for a group is a summary of the query that the results matched. More precisely, the header contains the (unqualified) type name and the field and method names used in the query. When query merging occurs, the header concatenates the summaries for all of the queries that were merged. Since page topics may not be clear from the page title alone, the header provides an important hint to the developer about which code elements the web pages discuss.

We also apply some special processing to deal with distinct URLs with near-identical content. In Reverb, this problem shows up frequently for online API documentation, where the same content can be hosted at separate locations, or multiple versions of a page can exist, corresponding to different versions of an API. We have implemented a couple of mechanisms to mitigate this issue. If two URLs have the same Lucene relevance score for a query, the same page title, and the same final URL segment, then the result with the higher frecency value is returned. If two URLs differ only in a single segment, and that segment appears to be a version number, the page corresponding to the higher version number is returned. A regular expression is used to identify segments that correspond to version numbers.

V. Reverb: Evaluation

Reverb is a recommender. The most common way to evaluate a recommender is using precision and recall: how many of the web pages returned by Reverb are applicable in a given situation (precision) and how many of the possible web pages that could be applicable were returned (recall). For Reverb, we were interested in precision over recall. Specifically, we were interested in whether Reverb can return one code-related page of interest within a set of recommendations presented to the developer as a developer is likely to stop using the web when an answer to their question of interest is found. We call a revisit that occurs a *hit* when it is one of the top 10 results suggested by Reverb when the revisit occurs. The *hit rate*—whether Reverb can return a web page of interest—is equal to the number of hits divided by the total number of code-related revisits initiated by the developer.

To investigate Reverb's hit rate, we conducted a field study of the tool in which participants had the tool installed during at least six hours of Java programming. Participants in the study could choose to have the Reverb view in which recommendations appear visible or not. If visible, we tracked how often developers took recommendations presented by Reverb. Given the difficulty of having developers use new tools, we also tracked the recommendations Reverb was making and the pages participants were visiting in their browsers. This information allowed us to track Reverb's blind hits: code-related revisits that were not initiated through Reverb, but which Reverb successfully predicted.

A. Study Setup

Each participant installed Reverb on their own computer. This version of Reverb logged all recommendations generated, web pages that were actually visited by a developer in their browser and whether visited pages were code-related or not based on the heuristics used in the code revisitation study (Section III). This version of Reverb also logged mouse and keyboard events in the Java code editor of Eclipse. If, in a 15-minute interval, a developer had at least one interaction in the Java code editor, we counted that interval towards the quota of use before a developer was accepted into our study. We tracked Reverb's performance until a quota of six hours of programming in Java had occurred for a participant. We chose six hours based on the rate of code revisitations in our earlier study: with one to two code revisits per hour on average, six hours would likely enable a reasonable number of revisitations to occur while respecting the time of our participants. In the study, it took participants between one and seven days to accumulate the six hours of coding activity.

At the half-way and end points of the study, participants were prompted to rate and comment on each recommendation on which they had clicked. The rating scale was from 1 to 5 with 1 labeled as "not useful" and 5 labeled as "very useful". At the end of the study period, participants were asked to complete a survey about their impressions of the tool and the ways it could be improved.

To allow participants to receive useful recommendations right away, we pre-populated their browsing history index using the same approach as taken in the page classification tool in our earlier study. The content of each page in the browser's history database for the three-months previous was downloaded and added to the index.

B. Participants

We advertised the study on Eclipse mailing lists, through contacts at various companies and within the UBC computer science department. To be eligible, participants had to be working on the Windows operating system[6] and be coding actively in Java using the Eclipse development environment. Ten participants completed the study. For one of these participants, his or her activity required 103 days to produce 6 hours of Java coding. We decided not to include this participant's data in our analysis as we deemed that they did not meet the requirements of coding actively in Java. The remaining nine participants for whom we analyzed data met the six hour quota of Java programming in a mean of 4 days $\pm$ 2.3.

C. Results

Table I summarizes the results of the field study. The first two rows, "View open %" and "Recommendations clicked", provide insight into the use of the Reverb recommendation view in Eclipse. Although the view was often open for five (55.5%) of the participants, few recommendations were clicked. Two of the three participants who completed comments about the tool noted that it was difficult to identify links from the page title alone, potentially causing them not to use the view more often. Designing an interface for presenting recommendations is difficult; as a result, we were not surprised that the number of recommendations clicked was low across participants. These difficulties might also have affected the user's ratings of recommendations. The average rating from four (44.4%) of the participants was just over three, indicating that the developer's saw the recommendations as potentially but not overwhelmingly useful. Given the issues developers had with the simple interface through which we provided recommendations, we focus the rest of our analysis of the data on Reverb's hit rate.

The second last row of Table I reports on the precision of Reverb's recommendations as Reverb was deployed to the participants. The row marked "Reverb Hit Rate, initial" reports on the precision of Reverb using the parameters as described in Section IV and includes both web pages whose visit was initiated through the Reverb view and whose visit occurred through the web browser, but at the same time that Reverb had made the recommendation. The hit rate reported is the average hit rate across all cases of Reverb making up to ten recommendations. The data shows that the mean hit rate for Reverb was 42% ($\pm$32.6%).

The final row marked "Reverb Hit Rate, optimized" reports on Reverb's precision when the tool is tuned. In the configuration provided to participants, Reverb generated a new set of recommendations each time a click event occurred in the Java editor, provided that the visible region in the code editor was different from the previous click. We say this implementation has a recommendation window of 10. However, since we logged all recommendations that Reverb generated, we could experiment with gathering recommendations over a varying number of recommendation windows. Specifically, we considered the recommendations that would have occurred given the last two visible code regions (i.e., window of 20) and the last three visible code regions (i.e., window of 30) and used our ranking engine to produce the top ten recommendations resulting from the different window sizes. We also wanted to tune the ranking parameters τ, the constant used in calculating the decay of the frecency boost and α, the geometric weighting of the relevance relative to frecency in the ranking function. Through simulations, we determined that the best hit rate was achieved with a recommendation window of 30 and a shorter time constant, corresponding to a half-life of 15 days (as opposed to the 6 months chosen initially). The final row shows that with these optimized settings, the mean precision for Reverb was 51% ($\pm$30.3%) and reached a high of 100% for one participant. This 51% hit rate is promising, suggesting that local code context can help to predict code-related page revisits.

D. Limitations

Our field study involved a limited number of participants. As accessing web data, even when anonymized is sensitive for

[6]This constraint was a result of the secure approach we took in connecting the browser to the development environment.

TABLE I
RESULTS OF FIELD STUDY OF REVERB

	Participant										
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**Average**	
View Open %	78	35	89	0	78	81	50	73	0	54	±34.76
# Recommendations Clicked	2	2	3	0	1	0	0	6	0	1.6	±2.01
Average Rating (out of 5)	2.5	3.5			3			3.5		3.13	±.48
Reverb hit rate, initial (in %)	25	57	22	17	100	0	31	83	43	42	±32.60
Reverb hit rate, optimized (in %)	50	57	22	33	100	0	81	67	50	51	±30.29

users, we chose to run the study as anonymously as possible. As a result, we are not able to report on the background of these developers. We also architected the tool to be highly secure resulting in constraints on the environments in which the tool could run (e.g., Windows and two popular browsers), leading to difficulties in recruiting participants to the study. The small numbers and potentially very diverse population of participants may have lead to the high values of standard deviation in the hit rates computed.

The classifier we used to determine if a page is code-related is not perfect. There may have been pages mis-characterized as code-related and pages mis-characterized as not code-related. Both kinds of mis-characterizations affect the hit rate we computed for Reverb.

Changes in the list of recommendations presented to developers using Reverb could affect their behavior. The 51% hit rate includes revisits that participants initiated through Reverb. Given a different set of recommendations based on tuned algorithm parameters, participants might not have clicked on the links that they did in the original study. To investigate this effect, we also computed the hit rate when revisits initiated through Reverb are ignored. Revisits initiated outside of the tool are unlikely to be affected by the recommendation set presented in the tool. When only these revisits are included, the optimized algorithm achieves a mean hit rate of 47% (± 32%).

It is important to note that the hit rate in the simulations we ran is limited by the data we collected in the field study. Only the recommendations that were actually presented to the user are included in the logs. So the simulation is limited to recommending pages that scored in the top 10, based on the ranking parameters initially chosen for the study. A higher hit rate might be achievable if the logs included a larger range of pages matching the generated queries.

VI. DISCUSSION

Is an average hit rate of 51% for the optimized version of Reverb sufficient to provide value to developers? This hit rate means that the Reverb view will often contain at least one web page of likely use to a developer at work and often that one page is sufficient to continue or complete the current work. For six (67%) developers in our study, the hit rate was over 50% with three (33%) developers having a hit rate over 67%. What might help improve the hit rate for a greater population of developers? We discuss possible improvements for Reverb's recommendation engine. We also discuss possible improvements to the presentation of results to the user.

Currently, Reverb looks for similarities in the content of code being worked on and web pages that have been visited. An alternative, similar to that taken in Codetrail [13], is to also incorporate temporal associations between code interactions and web page visits. In many cases, a content-based association may not exist, even though the page and the code are frequently visited together. A web page describing a particular design pattern or bug report (which does not mention particular code elements) are examples where the temporal association with source files or projects may be stronger than one based on content.

Developers did not click on many recommendations shown by Reverb in the field study. This low click through rate may be due to developers not having sufficient time to get used to the tool or it might be related to a lack of detail provided about the web pages recommended. Possible improvements to the existing user interface of Reverb include providing page snippets and/or thumbnails in the Reverb view. A more radical change would be to surface Reverb recommendations inside the browser, rather than in Eclipse. Although this strategy moves the recommendations away from the context to which they relate, it may increase the likelihood of developers using the recommendations. Or, if developers tend to alternate code-related refinding with searches for new content, it may also be beneficial to embed the recommendations in the browser.

A natural extension to Reverb would be the ability to access code-related pages other developers found useful. For example, a developer could flag a frequently-visited page through the tool, and then other members of her team would see that page at the top of their recommendations when working with related code. This approach might improve the hit rate, by softening the requirement that a page must have been previously visited before it can be recommended.

VII. SUMMARY

We have explored the problem of web page revisitation in the sphere of software development. Our formative study characterized how frequently developers return to code-related web pages and the methods they use to find these pages. We found that 13.7% (± 10.6%) of visits to code-related pages were revisits. Only a small fraction of these revisits (7.4%) were initiated through bookmarks, indicating that many code-related pages must be refound through more onerous means. Our dataset also contains many one-hour periods with three or more code-related revisits, suggesting that the number of revisits may rise during certain periods. Our study suggests that developers may benefit from revisitation support, particularly during certain coding activities.

To assist developers with code-related revisits, we introduced Reverb, which proactively recommends previously-visited web pages, based on the code currently visible in the developer's code editor. It extracts code elements from the code currently visible in the editor, constructs queries using the names of those elements, and runs those queries against a full text index of the developers browsing history. It combines cosine similarity with frequency and recency of page visits to determine the rank of recommendations, and aggregates results according to the code element queries they match.

To understand if Reverb can recommend appropriate web pages, we conducted a field study of Reverb. Our results suggest that local code context is indeed predictive of web page revisits, with an average 51% of code-related revisits being predicted by an optimized version of Reverbs ranking algorithm.

Acknowledgements

The authors would like to thank Microsoft for a Software Engineering Innovation award that helped fund this work, the individuals who were willing to install research prototypes in their work environments and participate in the studies on which we report and the anonymous reviewers of the submission for their helpful comments.

References

[1] J. Brandt, P. J. Guo, J. Lewenstein, M. Dontcheva, and S. R. Klemmer, "Two studies of opportunistic programming: interleaving web foraging, learning, and writing code," in *Proc. of the 27th Int'l Conf. on Human Factors in Computing Systems*. New York, NY, USA: ACM, 2009, pp. 1589–1598.

[2] M. Kersten and G. C. Murphy, "Using task context to improve programmer productivity," in *Proc. of the 14th ACM SIGSOFT Int'l Symp. on Foundations of Software Engineering*, ser. SIGSOFT '06/FSE-14. New York, NY, USA: ACM, 2006, pp. 1–11.

[3] N. Sawadsky and G. C. Murphy, "Fishtail: From task context to source code examples," in *Proc. of the 1st Workshop on Developing Tools as Plug-ins*. New York, NY, USA: ACM, 2011, pp. 48–51.

[4] H. Obendorf, H. Weinreich, E. Herder, and M. Mayer, "Web page revisitation revisited: Implications of a long-term click-stream study of browser usage," in *Proc. of the SIGCHI Conf. on Human Factors in Computing Systems*. New York, NY, USA: ACM, 2007, pp. 597–606.

[5] B. McKenzie and A. Cockburn, "An empirical analysis of web page revisitation," in *Proc. of the 34th Annual Hawaii Int'l Conf. on System Sciences (HICSS-34)-Volume 5 - Volume 5*. Washington, DC, USA: IEEE Computer Society, 2001, pp. 5019–.

[6] L. Tauscher and S. Greenberg, "How people revisit web pages: Empirical findings and implications for the design of history systems," *Int. J. Hum.-Comput. Stud.*, vol. 47, no. 1, pp. 97–137, Jul. 1997.

[7] E. Adar, J. Teevan, and S. T. Dumais, "Large scale analysis of web revisitation patterns," in *Proc. of the Twenty-sixth Annual SIGCHI Conf on Human Factors in Computing Systems*. New York, NY, USA: ACM, 2008, pp. 1197–1206.

[8] S. Greenberg and A. Cockburn, "Getting back to back: Alternate behaviours for a web browser's back button," in *Proc. of the 5th Annual Human Factors and the Web Conf.*, 1999.

[9] N. Milic-Frayling, R. Jones, K. Rodden, G. Smyth, A. Blackwell, and R. Sommerer, "Smartback: Supporting users in back navigation," in *Proc. of the 13th Int'l Conf. on World Wide Web*. New York, NY, USA: ACM, 2004, pp. 63–71.

[10] R. Kawase, G. Papadakis, E. Herder, and W. Nejdl, "Beyond the usual suspects: Context-aware revisitation support," in *Proc. of the 22nd ACM Conf. on Hypertext and Hypermedia*. New York, NY, USA: ACM, 2011, pp. 27–36.

[11] J. Stylos and B. A. Myers, "Mica: A web-search tool for finding api components and examples," in *Proc. of the Visual Languages and Human-Centric Computing*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 195–202.

[12] R. Hoffmann, J. Fogarty, and D. S. Weld, "Assieme: Finding and leveraging implicit references in a web search interface for programmers," in *Proc. of the 20th Annual ACM Symp. on User Interface Software and Technology*. New York, NY, USA: ACM, 2007, pp. 13–22.

[13] M. Goldman and R. C. Miller, "Codetrail: Connecting source code and web resources," *J. Vis. Lang. Comput.*, vol. 20, no. 4, pp. 223–235, Aug. 2009.

[14] B. Hartmann, M. Dhillon, and M. K. Chan, "Hypersource: Bridging the gap between source and code-related web sites," in *Proc. of the 2011 Int'l Conf. on Human Factors in Computing Systems*. New York, NY, USA: ACM, 2011, pp. 2207–2210.

[15] J. Brandt, M. Dontcheva, M. Weskamp, and S. R. Klemmer, "Example-centric programming: Integrating web search into the development environment," in *Proc. of the 28th Int'l Conf. on Human Factors in Computing Systems*. New York, NY, USA: ACM, 2010, pp. 513–522.

[16] A. Bacchelli, M. Lanza, and R. Robbes, "Linking e-mails and source code artifacts," in *Proc. of the 32nd ACM/IEEE International Conf. on Software Engineering - Volume 1*. New York, NY, USA: ACM, 2010, pp. 375–384.

[17] N. Sawadsky, "Reverb: Dynamic bookmarks for software developers," M.Sc. Thesis, Dept. of Computer Science, University of British Columbia, 2012.

[18] S. Bajracharya, T. Ngo, E. Linstead, Y. Dou, P. Rigor, P. Baldi, and C. Lopes, "Sourcerer: A search engine for open source code supporting structure-based search," in *Companion to the 21st ACM SIGPLAN Symp. on Object-oriented Programming Systems, Languages, and Applications*. New York, NY, USA: ACM, 2006, pp. 681–682.

[19] G. Cormode, V. Shkapenyuk, D. Srivastava, and B. Xu, "Forward decay: A practical time decay model for streaming systems," in *Proc. of the 2009 IEEE Int'l Conf. on Data Engineering*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 138–149.