REVERB: DYNAMIC BOOKMARKS FOR DEVELOPERS

by

Nicholas Sawadsky

B.A.Sc., Simon Fraser University, 1995

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in

THE FACULTY OF GRADUATE STUDIES

(Computer Science)

THE UNIVERSITY OF BRITISH COLUMBIA

(Vancouver)

July 2012

Abstract

The web is an increasingly important source of development-related resources, such as code examples, tutorials, and API documentation. Yet existing integrated development environments do little to assist the developer in finding and utilizing these resources. In this work, we explore how to provide useful web page recommendations to developers by focusing on the problem of refinding previously-visited web pages. We present the results of a formative study, in which we measured how often developers return to code-related web pages, and the methods they use to find those pages. Considering only revisits which occurred at least 15 minutes after the previous visit, and are therefore unlikely to be a consequence of browsing search results, we found a code-related recurrence rate of 13.7%. Only 7.4% of these code-related revisits were initiated through a bookmark of some kind, indicating the majority involved some manual effort to refind. To assist developers with code-related revisits, we developed Reverb, a tool which displays a list of dynamic bookmarks that pertain to the code visible in the editor. Reverb's bookmarks are generated by building queries from the classes and methods referenced in the local code context and running these queries against a full-text index of the developer's browsing history, as collected from popular browsers used. We describe Reverb's implementation and present results from a study in which developers used Reverb while working on their own coding tasks. Our results suggest that local code context can help in making useful recommendations.

Preface

The two user studies described in this thesis were approved by the UBC Behavioural Research Ethics Board under certificates H11-01982 and H11-01982-A001.

Table of Contents

List of Tables

List of Figures

Acknowledgements

I am grateful to my wife, Anita Parkinson, for encouraging me to pursue a Master's, and giving me so much support through the whole process.

My heartfelt appreciation goes to Dr. Gail Murphy for encouraging me to apply to UBC, guiding me towards an interesting, multi-faceted thesis topic, and asking lots of hard questions along the way.

Many thanks to the students and developers who took the time to participate in the two user studies documented here. Also thanks to Nathan Hapke for his work in building the original Fishtail prototype, out of which Reverb emerged.

Finally, I would like to acknowledge the financial support of Microsoft Corporation in funding portions of this research.

Chapter 1: Introduction

The web plays an increasingly important role as an information resource for software developers. In addition to formal API documentation, the web contains a wealth of tutorials and example code to help developers as they begin to work with a new library or framework. The specific set of method calls and parameters needed to achieve a given API behavior can often be found with a well-chosen set of search keywords.

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 particular, the retrieval of relevant web resources is an effortful process, requiring the developer to switch away from their development environment to the browser, choose a set of search keywords, enter them, and then evaluate the results. The process is often iterative, as the developer refines keywords through a series of searches.

In this thesis, we are concerned with streamlining the retrieval of relevant web resources for developers. Our vision is of a tool which automatically predicts links that the developer will find useful as she works. If accurate predictions can be made with high enough frequency, such a tool could dramatically reduce the cognitive and manual effort consumed in web search.

In an initial proposal [28], we outlined a tool called Fishtail which automatically constructed queries from the code under development, ran those queries on a general-purpose web search engine, and displayed results within the Eclipse IDE. Informal experiments with this approach showed some limitations. The results returned did not appear as useful as we had hoped. Fishtail's automatically-constructed queries did find examples and documentation relating to the libraries referenced by the code, but the results were not focused on the facets of those libraries that were of greatest interest to the developer.

The Fishtail experiment led us to seek contextual clues to the aspects of the code that the developer needed most assistance with. We settled on the developer's web browsing history as a promising and under-utilized information source. Our basic idea is to limit the search to pages the developer has previously visited. By doing this, we help ensure that results are

focused on areas that the developer found unclear at some point in the past. In addition, we reuse the developer's previous cognitive effort in choosing the most useful pages from the result list presented by the search engine.

By limiting the search to previously-visited pages, we hope to improve the ability of the tool to predict page revisits. However, this comes at a significant cost: the tool can no longer recommend visits to new pages that have never been visited before. How useful can a recommendation tool be, if it only recommends pages the developer has already seen?

To answer this question, we ran a formative study with 11 developers. In this study, the developers ran a program which analyzed their web browsing history. The program automatically classified pages as code-related or non-code-related. For code-related page visits, we were then able to calculate the recurrence rate: the percentage of all page visits that were revisits. Using a 15-minute filter (ignoring revisits that occurred less than 15 minutes after the previous visit), we found a mean code-related recurrence rate of 13.7%. The 15-minute filter eliminates short-term revisits that tend to be initiated through a link from a page that is already open, such as a search results page or a root page within a site.

Our data also suggest that there may be development activities during which page revisits become more frequent. Five of eleven participants in our study had at least three one-hour periods containing three or more code-related revisits. For two participants, the number of these periods was nine or greater. The actual number of periods containing significant revisit activity (as well as the number of revisits in these periods) is likely higher still, given the limited recall of the classifier used to identify code-related pages.

Echoing previous studies, we found that the majority of revisits were through a link from another page, rather than through bookmarks or address bar autocomplete. Refinding through repeated queries or retracing of previous navigation paths requires both manual and cognitive effort. While not conclusive, our study suggests that a tool which can accurately anticipate revisits could yield meaningful effort savings.

To explore the potential of code-related revisitation support, we developed a tool called Reverb. Reverb includes browser extensions, an indexing service, and an Eclipse plugin. As

the developer browses the web, page content is added to an index. Then as she works on code within the IDE, related web pages from her browsing history are displayed in a view alongside the code. Reverb shares similarities with other research-based tools such as Codetrail [10] and HyperSource [13], which seek to link source code with web pages the developer has visited. However, Codetrail generates links to general web pages only when a very close match between source code and page content is detected, while HyperSource relies on temporal locality of browsing and code editing. Reverb uses keyword-based matching, which can retrieve a broader range of pages connected to the current source code.

We conducted a small field study of Reverb. Three developers participated, using the tool in their own development tasks for a period of six hours. We gathered participants' qualitative impressions of the tool and suggestions for ways it could be improved. We also collected quantitative data on how many recommendations participants clicked on and how well recommendations correlated with the pages they actually visited. Our initial results suggest that local code context is indeed predictive of web page revisits, with 35% of code-related revisits appearing in Reverb's top 10 recommendations. The hit rate reaches 57% when we run an optimized version of the ranking algorithm on the data collected in the field study.

The key contributions of this thesis are:

a) empirical measurements of how often developers return to code-related web pages and the methods they use to perform these revisits;
b) a description of a novel tool to support developers in refinding code-related web pages, by proactively recommending pages from their browsing history that relate to the code under development; and
c) the results of a user study of this tool, which suggest that local code context is helpful in recommending useful web pages.

Chapter 2: Related Work

The related work can be divided into four broad categories:

a) empirical studies on web page revisitation;
b) revisitation support for general web pages;
c) content-based contextual recommenders; and
d) tools that support developers in their use of the web.

2.1 Web Page Revisitation Studies

An important early study focused on web page revisitation is that of Tauscher and Greenberg from 1995 [31]. They introduced the recurrence rate metric, which measures the likelihood that a given page visit is a revisit. They also pioneered the approach of collecting data from the user's web browser, rather than relying on server logs to analyze navigation patterns. Participants ran a modified version of the XMosaic browser over a period of six weeks. The reported recurrence rate for this study was 58%.

In 1999, McKenzie and Cockburn calculated a recurrence rate of 81%, based on the browsing history files of 17 Netscape users. The history files were gathered from incremental backups spanning a period of four months [22]. The relatively high recurrence rate may have been partly due to a data preprocessing step which truncated URLs, ignoring query parameters.

In 2004-2005, Obendorf et al. conducted a long-term study with 25 participants using the Firefox browser [25]. Instead of relying on the history database, an intermediary was installed on participants' computers which recorded all page requests. Data was collected over spans ranging from two to seven months. The authors argue for a definition of a distinct URL which includes the query parameters and POST data, and calculate a recurrence rate of 45.6%, lower than that measured in earlier studies.

In 2008, Adar et al. published a revisitation study which drew on a very large user base [1]. The data set consisted of anonymized browsing information gathered from 612,000 users of the Windows Live Toolbar over a five-week period in 2006. While earlier studies characterized the revisitation behavior of users across all pages they visited, their study

attempts to identify revisitation patterns associated with particular web pages. They calculate an average revisitation curve across all pages, and then cluster pages in to fast, medium, slow, and hybrid groups, depending on how a page's revisitation curve compares to the centroid.

Our formative study on developer revisitation patterns is modeled on that of Obendorf et al. However, their work is mainly concerned with broad web revisitation patterns. In our attempt to discern differences in revisitation patterns between code-related and non-code-related pages, our study shares some similarities with the work of Adar et al. Where they begin by learning a number of revisitation curves, and then drill down to characterize the pages matching those curves, we start from a set of pages classified as code-related, and then determine the associated revisitation profile.

While it makes no mention of revisitation, the informal study conducted by Goldman and Miller [10] is notable in focusing on the code-related web browsing behavior of software developers. Through manual classification, they determined the percentage of participants' page visits which were code-related (10%). They also divided code-related pages that participants visited in to categories such as formal API documentation (30%), technical resources such as articles, blogs and tutorials (24%), searches for specific code elements (12%), and searches for code concepts (11%). Since this study, other researchers have characterized in more detail the types of searches developers do while engaged in programming tasks [14,29]. Most relevant to our work, Brandt et al. studied query logs from users of the Adobe Developer Network web site, finding that 22% of sessions had "learning traits," while 78% had "reminding" traits [6]. Their results suggest that developers frequently perform searches to remind themselves of the details of API behavior. However, the accuracy of the heuristics used to identify sessions with reminding traits was not assessed, and they do not report what fraction of such sessions actually led to a page revisit.

2.2 Revisitation Support

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. Some have proposed extending the functionality of the back button [8,23], or enriching browser history

with visual and contextual cues [12,16]. Others have explored clustering items in the browsing history using site structure, link structure, content similarity, and visit time [18,30]. While many researchers note the dominance of the recency effect in determining which pages will be revisited, some have also been intrigued by the recurrence of specific navigation trails in users' browsing histories [7,24]. For example, a visit to a user's web-based email client may frequently be followed by a visit to her online calendar. A recent study found that augmenting a frequency- and recency-based ranking algorithm with page transition probabilities from the user's browsing history increased the percentage of successful revisit predictions from 71.7% to 81.8% [17]. In a user study of PivotBar, a browser extension based on this layered approach, users initiated 22.7% of revisits through the extension.

Similar to PivotBar, Reverb uses contextual information to predict pages that the user may wish to revisit. However, the context considered is different. Where PivotBar associates the current web page with previous navigation trails, Reverb relates the content of the developer's code editor with similar web pages in the user's browsing history.

2.3 Content-based Contextual Recommenders

Research interest in content-based contextual recommendation tools dates from the late 90s, when a variety of systems were proposed for just-in-time information retrieval (JITIR) and web reconnaissance [19,26]. The key features of these agents are the proactive presentation of suggestions to the user and the use of some notion of local context in making these suggestions. Local context typically includes the content of the current document or web page being viewed and can be augmented with a keyword-based user profile based on a history of the user's activities. Recommendations can be drawn from a document repository (such as an email archive or a database of paper abstracts), the web at large, or the local link neighborhood of the current web page. Matching of local context with items in the repository is performed using standard information retrieval (IR) techniques, such as cosine similarity with term-frequency/inverse-document-frequency (tf/idf) normalization.

Source code is one domain of information that can be used as context for recommendations. For instance, systems have been proposed which use the local code context to recommend

specific API elements from software libraries [32] or source code snippets mined from repositories [15]. More recently, researchers have devised contextual recommenders which focus specifically on the task of object instantiation [2,21,27]. These systems use structural attributes to match code in the editor with extracts from a repository.

Contextual recommendation systems face a number of challenges. It can be difficult to discern the user's near-term intent from the local context alone. And the local context may point in various directions. For example, the current method may reference a number of classes, but perhaps only one of these will lead to contextual recommendations that the user finds helpful. A trade-off arises between the precision of recommendations and the goal of minimizing user effort.

The approach taken in Reverb is to limit the scope of the recommendation pool to web pages that the user has previously viewed. By recommending only items in which the user has already expressed an interest, the system can improve the likelihood of making useful recommendations. A patent filed in 2003 describes a system for automatically indexing visited content and later making proactive recommendations for these items based on the current context [4], but, to our knowledge, no practical implementation or evaluation of such a system has been described.

2.4 Tools to Support Developers' Use of the Web

A variety of tools have been proposed to assist developers in their use of the web. Mica [29] and Assieme [14] are examples of interfaces which facilitate searching for code-related resources on the web. These tools categorize search results by the API's and types that they reference. They allow a developer starting from a task-oriented query (e.g., "java create zipfile") to quickly identify API's and code examples relevant to the task.

Codetrail [10] creates a communication link between the browser and the IDE. As the developer browses the web, Javadoc and other well-structured API documentation is automatically detected and linked to source code. Reverb's IR-based approach achieves the same benefit but also works well for less-structured forms of documentation. In addition, Codetrail can detect blocks of code on visited web pages which match code in the

developer's workspace, and automatically generate a bookmark in the workspace to capture the link. This approach requires a close match between code on the web page and code in the workspace. By relying on automatically-generated queries rather than sequence matching, Reverb can link code to related web pages which do not necessarily contain blocks of matching code.

HyperSource [13] maps web pages visited to corresponding source code edits and maintains these associations over time. 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.

Blueprint [5] embeds the search interface in to the IDE, overlaying the search box and result list atop the source code editor. It does not provide results proactively, but does use local context to augment the user-entered query (adding, for example, the name of the programming language). While Blueprint is not specifically focused on revisitation support, an increase in repeat queries was measured for users equipped with the tool.

Chapter 3: How Many Visits are Revisits?

In our initial study, we wanted to explore how frequently developers revisited code-related web pages, and the methods they used to refind previously-visited pages. A key metric in previous studies is the *recurrence rate*, defined in [31] as:

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

The recurrence rate gives the probability that a given page visit is a revisit.

In the long-term click-stream study of Obendorf et al. the average recurrence rate was 45.6% [25]. This recurrence rate was calculated across all the pages visited by the participants, including pages that would typically be visited many times a day, such as the browser home page, web search engines, and web-based email applications. We knew that the recurrence rate for code-related web pages would be lower, but we believed that still a substantial portion of code-related visits would be revisits. In addition, we believed that the intervals between code-related revisits would be longer than for non-code-related revisits. Revisits that are further apart in time may require more developer effort, since it will be more difficult to recall the navigation path or set of query keywords originally used to retrieve the page.

We also wanted to determine the mechanisms developers used to revisit web pages. Previous studies have shown that bookmarks play only a minor role in users' revisitation strategies. In [25], only 13.2% of revisits were initiated through "direct access," a category which included bookmarks, the home page button, the history list, and typed URLs. In that study, the browser back button and clicking a link on another page (e.g., a search results page) were the most common revisitation mechanisms, at 30.9% and 29.6% respectively. We wanted to confirm that similar rates also held in the case of code-related web browsing.

3.1 Study Setup

Similar to the study of McKenzie and Cockburn [22], our approach was to make use of the detailed browsing history databases maintained by the Mozilla Firefox and Google Chrome

web browsers. Both browsers use the SQLite database engine[1] for managing browsing history. Each time the user visits a page, an entry containing the timestamp and a location ID is added to the 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.

An advantage of using existing browsing history databases is that a large amount of history data can be collected while still requiring only a short time commitment (approximately 15 minutes) on the part of study participants. In addition, since participants are only asked to provide their data after it has been generated, we avoid the risk of users changing their behavior because they know they are participating in a study.

We wrote a tool which participants could run on their computers at a time convenient for them. The tool prompted participants to choose a browsing history window of at least two months that included periods of active coding. The default window was from the date when the participant ran the tool to three months earlier. The average browsing history window selected was 82 days, and a total of 906 days of browsing history was collected.

The tool extracted a list of web page visits 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. The jsoup HTML parser library[2] was used to download and process the HTML of the pages. `<script>` elements were first filtered from the page, as these elements are not displayed to the user and can contain tokens which would make a page appear code-related.

The text content of the page was then extracted and the page was classified as code-related or non-code-related, using heuristics discussed 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 containing the anonymized browsing history, the page

[1] See http://sqlite.org/.

[2] See http://jsoup.org/.

classifications, and the manual classifications performed by the participant. Anonymization was performed by replacing the URL of each page visit with the numeric identifier assigned to that location in the browser history database. The source code for the tool used by participants in this study is available on Google Project Hosting.[3]

We recruited participants from within the UBC computer science department and from one software development company. Eight participants were graduate students and three were professional software developers. Seven participants used the Google Chrome web browser and four used Mozilla Firefox. While Java was the programming language used by the largest number of participants (seven), a range of other programming languages was represented: C, C++, Javascript, Perl, Ruby, PHP and Python. Participants ran the study tool between September and November 2011.

3.2 Classifying Code-Related Pages

Accurate classification of web pages as code-related or non-code-related poses a significant challenge. The main types of resources we had in mind were code examples, tutorials, and formal API documentation. One approach for identifying such pages would be to search for blocks of text which appear to be code, based on characteristics such as the density of delimiter tokens (e.g., ";", "{", "}") and the presence of keywords from a particular programming language. This approach would need to be customized for each language to be supported. It would also easily miss pages that do not contain multi-line code examples – much API documentation would fall into this category.

The approach above would also fail to identify pages which deal with high-level programming concepts (e.g., concurrency) but contain no specific code examples. To capture such pages, an approach such as latent semantic indexing (LSI), which can identify clusters of keywords representing code-related topics, seems more appropriate. Before LSI can be applied, the indexer has to be trained on a large corpus of documents. Training the indexer is computationally intensive, and the resulting topic model is likely to be quite specific to the

[3] See http://code.google.com/p/reverb-bookmarks/.

corpus on which it is based. In addition, in a recent study of techniques for linking email messages with the source code they discuss, Bacchelli et al. found that LSI performed well on recall, but fared poorly in the precision of its results [3].

A third approach was based on the camelCase naming conventions prevalent in current coding practice. The idea here was to construct a regular expression that would match terms containing a medial capital letter. Medial capitals (capitals that are not the first letter in the word) occur frequently in class and method names, but are less common in non-code-related text. This approach had the advantage of being relatively independent of programming language, provided the API being discussed followed a camelCase naming convention. Multi-line code examples were not required, so pages which contained mainly descriptive text with only a few mentions of class or method names could still be flagged. And the effectiveness of the approach could be estimated quickly using our own browsing histories.

Like the first approach, this third option would not be effective in detecting pages which deal with high-level programming concepts, without ever mentioning specific classes and methods. This highlights an important consideration in choosing between the three options: the relative importance of precision and recall. We believed that the third approach could be made quite precise. By adjusting our regular expressions, the false positive rate could be made very low, if we were willing to sacrifice recall. In order to estimate the recurrence rate for code-related pages accurately, high precision was required, but recall was less critical. High precision would also allow us to establish accurate lower bounds on the number of code-related visits and revisits made by our participants.

Based on the various advantages cited above, we chose to pursue the third option. Our initial regular expression searched for words which contained an underscore or looked like an identifier in camelCase. Our goal with this filter was to detect web pages containing type declarations and references and method declarations and invocations. In addition to camelCase, the other naming convention that is common for variable names and method names is to separate words in the name with underscore characters. By searching for camelCase patterns as well as identifiers containing underscores, we hoped to achieve broad coverage across programming languages that are currently popular.

In more detail, the regular expression matched words which:

a) contain one or more underscores; or
b) start with a lower-case letter and contain at least one upper-case letter; or
c) start with an upper-case letter and contain at least one lower-case, followed by an upper-case letter; or
d) start with two or more upper-case and contain at least one lower-case letter.

The regular expression was designed to avoid matching ordinary capitalized words and acronyms. A single upper-case letter at the start of a word was not sufficient to flag it as camelCase, nor was a word consisting of all upper-case or all lower-case letters considered a match. It is important to note that these restrictions came at a cost in recall. Many class names and method names (e.g., `Socket`, `File`, `File.delete()`, and `File.exists()` from the JDK) would not meet these restrictions, and web pages which referenced only such classes and methods would fail to be classified as code-related.

Even though it was constructed to avoid matching ordinary capitalized words and acronyms, the identifier filter described above generated many false positives. As it turns out, medial capitals occur fairly often in non-code-related web pages. For example, company names and product names can match this filter. To avoid these false positives, we extended the identifier filter to match only patterns that resembled method declarations or invocations with parameters. This pattern consisted of an instance of the identifier pattern, followed by an open bracket, at most 40 non-close-bracket characters, and a second identifier.

Restricting our 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, the pattern that looked for empty method invocations could be looser in its definition of what constituted a method name. CamelCase or the presence of an underscore was not required: any word beginning with a letter was a candidate to match this pattern.

At least two matches to these two patterns (method declaration/invocation with parameters and method declaration/invocation with no parameters) had to be found for a web page to be classified as code-related.

3.3 Results

Obendorf et al. highlight the importance of defining what is meant by a visit and a revisit, and point out that the definitions varied in earlier studies [25]. In our study, a page was identified by the 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, except as specified below.

As compared with an intermediary-based approach, relying on the browser's history database reduces the need for data preprocessing. Firefox and Chrome do not add separate history entries for loading of inline frames or automatically-loaded sub-frames within a frameset. Manual loading of a new page in to a sub-frame does result in a new history entry, and we counted these as separate visits. We filtered redirects, so that only a single visit was counted in each redirect chain. Beyond this, we did not do any filtering.

Table 1 summarizes participant attributes and the key results of the study. The overall recurrence rate averaged across all participants was 41% (σ = 14%). For just those pages flagged as code-related, the recurrence rate was 23% (σ = 12%). The 41% value is quite close to the 45.6% recurrence rate reported by Obendorf et al. Their value included forward and back button transitions, which our measurement excludes.

Across participants, the code-related recurrence rate varied widely, from a low of 11% to a high of 57%. For web pages in general, the recurrence rate ranged from 20% to 65%. This variability accords with previous studies. The recurrence rate seems to depend strongly on individual web browsing patterns, and the activities the participant pursued during the period for which data was collected.

Table 1. Summary of data from code-related revisitation study.

Participant	1	2	3	4	5	6	7	8	9	10	11	Mean
Occupation	Graduate Student	Graduate Student	Graduate Student	Graduate Student	Software Dev.	Software Dev.	Graduate Student	Graduate Student	Graduate Student	Graduate Student	Software Dev.	
Programming languages	C, C++, Java	Javascript, PHP	Java	Ruby, Perl	Java, C++	Java	Java	Java	C++	Python, Javascript	Java	
Web browser	Chrome	Chrome	Firefox	Chrome	Firefox	Chrome	Firefox	Chrome	Chrome	Chrome	Firefox	
History span (days)	23	91	93	91	73	72	151	89	91	84	48	82
Page visits	3281	8294	2853	18057	7672	14511	37068	15390	6144	10810	2286	11488
Distinct URLs	2210	3522	1354	12285	3632	4336	21744	5924	4032	6297	1190	6048
Overall recurrence rate	20%	50%	45%	32%	49%	65%	38%	58%	20%	37%	38%	41%
Recur. rate > 15 min	12%	27%	32%	22%	33%	47%	21%	42%	12%	25%	27%	27%
Code-related visits	129	220	35	613	107	385	392	292	56	405	114	250
Code-related URLs	103	158	28	453	91	316	308	127	50	320	97	186
Code-related URL %	4.7%	4.5%	2.1%	3.7%	2.5%	7.3%	1.4%	2.1%	1.2%	5.1%	8.2%	3.9%
Code-related recur. rate	20%	28%	20%	26%	15%	18%	21%	57%	11%	21%	15%	23%
CR recur. rate > 15 min.	12.7%	14.6%	0.0%	20.1%	8.1%	11.7%	13.5%	41.2%	5.7%	15.8%	7.6%	13.7%
Hrs with 3+ CR revisits	1	2	0	9	0	3	5	3	0	11	0	3.1
Classifier accuracy	100%	88%	88%	88%	96%	100%	100%	80%	88%	80%	100%	92%

Figure 1 depicts revisits bucketed according to the time since the previous visit, for code-related pages and for those classified as non-code-related. The revisit interval is measured relative to the most recent visit to the page. The figure shows that the revisit intervals for code-related pages do tend to be longer than for non-code-related pages. This effect is most noticeable below 15 minutes (51% for non-code-related versus 41% for code-related), and in the 10-30 day range (4% and 7%, respectively).

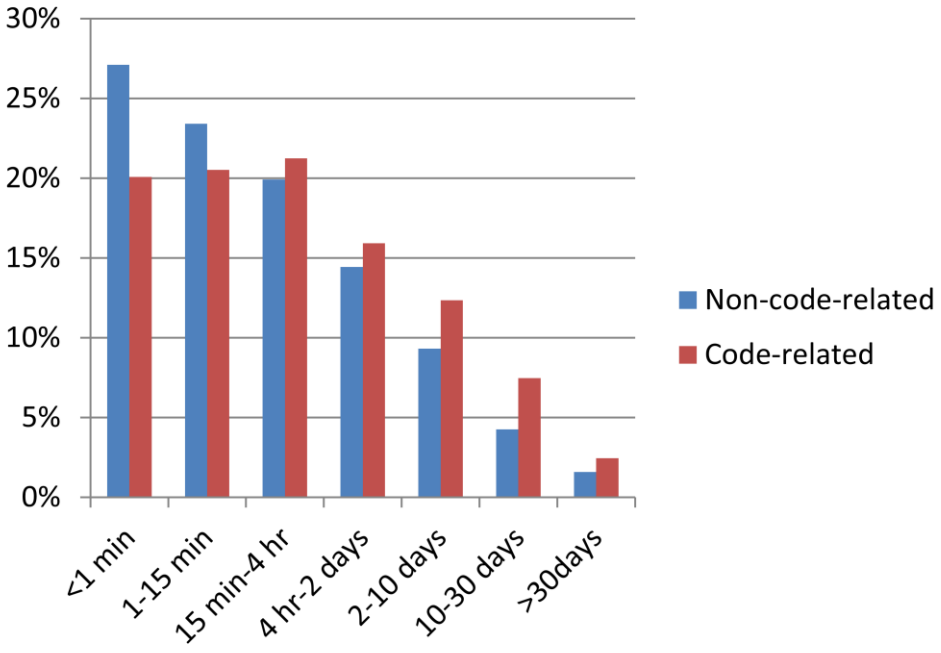

Figure 1. Revisits bucketed by time since previous visit.

While revisit intervals tend to be longer for code-related web pages, the number of code-related revisits occurring less than 15 minutes after the previous visit is substantial. This is an important factor when we consider tool support. Back-button revisits are omitted in these counts, but many of these short-term revisits are performed using links from pages the user already has open. For example, the user may repeatedly navigate to the same root page within a site, using the site's internal navigation mechanisms. Or from a root page or a search results page, the user may reopen a recently-visited page. We believe that a new revisitation support tool is unlikely to be used for short-term revisits initiated in these ways. In estimating the potential benefit of such a tool, we should focus on the revisits for which it is likely to be used. Choosing an appropriate time threshold is difficult. We judge that revisits longer than

15 minutes apart are likely candidates for additional tool support. At the 15 minute mark, even if the page is reachable from a page that is currently open, the user may have trouble remembering the page, or locating it in their browser tabs.

Accordingly, we report in Table 1 the code-related recurrence rate for revisits more than 15 minutes apart. In this calculation, revisits occurring less than 15 minutes after the previous visit are ignored: they are not counted in the numerator or the denominator of the recurrence rate. With this 15-minute window, the code-related recurrence rate is 13.7% ($\sigma = 10.6\%$), and the overall recurrence rate is 27.3% ($\sigma = 11.0\%$).

To estimate the accuracy of our page classifier, each participant was presented with 25 pages classified as code-related, and asked to flag any page that had been misclassified. The reported accuracy, or precision, of the classifier was high, averaging 92% ($\sigma = 8\%$). This was expected – as discussed above, we had tuned the classifier carefully on our own browsing histories to ensure high precision. We did not attempt to gather any measure of recall, but we expect it was low. This is reflected in the low percentage of page locations which were classified as code-related (3.9%). It is important to bear in mind the limited recall of the classifier when considering our results. The "non-code-related" pages likely contain a substantial number which would be considered code-related by any human classifier.

We also categorized code-related revisits according to the user action leading to the revisit. Figure 2 shows this breakdown. In this analysis, we again considered only revisits outside the 15-minute window. Because the Chrome history database provides a more fine-grained categorization of page transition types, only the data sets from participants using Chrome are represented in this graph. Unlike Firefox, Chrome flags visits that are initiated through the forward or back buttons, and we include those visits under the *Fwd/Back* category in this graph. The *Bookmark* category represents visits initiated through a bookmark in the browser, and includes bookmarks created automatically, such as the frequently-visited locations shown on Chrome's "New Tab" page. The *Start Page* category captures visits where the page URL was passed on the command line of the browser. This is the case, for example, when the user clicks on a bookmark on the desktop, or the browser is launched by another application, such as an IDE. The *Submit* category represents visits that were initiated through

a form submission. Visits initiated by typing in the address bar fall under the *Typed* category. Cases where the user clicked on a completion suggestion are included in this category. The *Link* category captures all visits initiated by clicking a link on another web page, including a search results page.

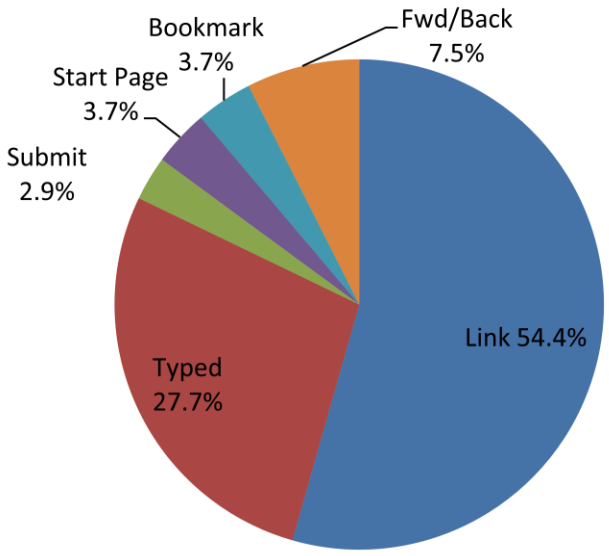

Figure 2. User actions leading to CR revisits > 15 min.

Notable in this chart is the small percentage of revisits (3.7%) performed using bookmarks. This is compelling evidence for the limited use of bookmarks in revisiting code-related web pages. At 54.4%, the dominant mechanism 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.). In other cases, these will be pages in a navigation trail the user follows to refind the page (*re-tracing*). 27.7% of code-related revisits fell in to the *Typed* category, indicating that address bar auto-complete is an important tool for developers in refinding web pages. Only 3.7% of code-related revisits fell under Chrome's *Start Page* category.

Figure 3 shows user actions leading to non-code-related revisits, again limited to participants using Chrome. The breakdown is similar, with links being used slightly more frequently

(59.0%). The higher percentage of revisits falling in the Submit category (8.2% versus 2.9%) is likely due to repeated search engine queries. Compared to code-related pages, non-code-related pages have a smaller percentage of forward/back button revisits. "Hub-and-spoke navigation" describes a pattern where the user returns to a hub page and navigates out to multiple spoke pages. Hub-and-spoke navigation within code-related sites (e.g., formal API documentation) may account for the larger percentage of forward/back button revisits in this category.

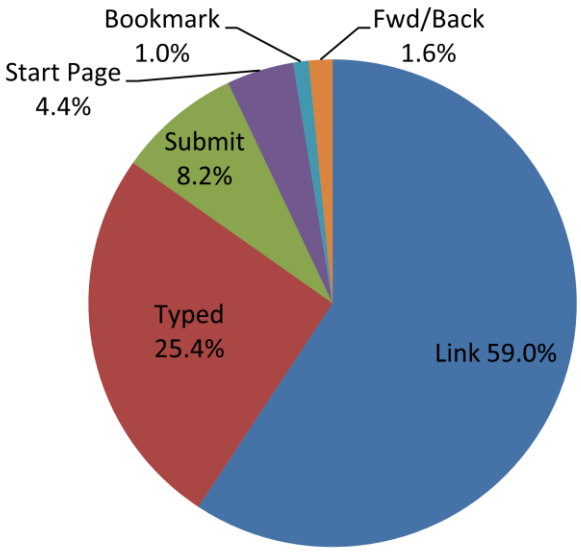

Figure 3. User actions leading to non-CR revisits > 15 min.

We also measured the number of one-hour periods in which at least three code-related revisits were observed. The 15-minute window was again used to filter these revisits. This value varied widely from a low of zero to a high of eleven for one participant. For five participants, the count of these periods was three or greater. For two participants, the count was nine or greater. As discussed earlier, our classifier was optimized for precision rather than recall. As a result, there will be many code-related visits which are not classified as such. Consequently, the counts of periods containing at least three code-related revisits provide a baseline, with the actual number of periods containing significant code-related revisit activity (as well as the number of revisits in these periods) being higher.

3.4 The Case for Tool Support

Using the 15-minute filter, the code-related recurrence rate is 13.7%, roughly half the value of 27% for web pages in general. The page transition data from our Chrome-based participants shows that only about 7.4% of code-related revisits outside the 15-minute window are initiated directly, through a bookmark inside or outside the browser. So the large majority of these revisits involve more manual effort than simply clicking on a bookmark. The developer must open her web browser and type a few characters of the URL or page title in to the address bar to bring up a completion suggestion. Or she must perform a refinding search using a web search engine. A refinding search can be time-consuming if she cannot remember the original search terms used, and must try a few different queries.

The frequency of code-related revisits is also important to consider. For five participants, there were three or more one-hour periods containing at least three code-related revisits. This suggests that there may be specific coding activities during which revisits are more frequent, and a support tool would be found helpful. It also appears that some developers revisit code-related web pages more frequently than others, and would stand to gain more from such a tool.

An additional factor in our analysis was the potential for small changes in effort to result in large changes in behavior [11]. Given an easier way to quickly revisit API documentation, technical articles, and code examples, developers may do so more frequently, and rely less on memory. This could have a benefit in improved code quality, and in developers' confidence in code correctness.

Based on these considerations, we felt that a tool to help developers refind code-related web pages was worth exploring further.

3.5 Limitations of the Study

As described in section 3.1, the tool for the study extracted a list of previously-visited web pages from the history database of the participant's web browser. It then downloaded the HTML content of each page using the jsoup parsing library, and classified each page as code-related or non-code-related. The approach of downloading page content outside the browser,

potentially months after the page was actually visited, has some limitations. The page content may have changed in the period since the last visit. If the page contains content which is generated dynamically by scripts, the dynamic content will not be loaded. If the page content is determined not just by the URL, but also by the access context (e.g., client-side cookies or POST parameters), then the page downloaded by the tool can differ from the page the participant actually viewed. If access control mechanisms are used to protect the page, the original page content will not be retrievable. The overall effect of these limitations is to reduce the number of pages classified as code-related, relative to the true number. We believe these limitations are unlikely to have a significant effect on the calculated recurrence rate.

Both Firefox and Chrome add a new entry in the browsing history when the user clicks on an anchor to navigate to a location within the same page. The URL for the visit is the base URL for the page combined with a hash character and the anchor identifier (the "fragment identifier"). In our results, these in-page navigation events are treated as distinct page visits. One effect of this is to boost the total number of code-related page visits and revisits. Because the anchors clicked will vary from visit to visit, the effect on the code-related recurrence rate is likely small.

Search result pages can be classified as code-related, if page titles or snippets match the regular expressions described earlier. Including search result pages in our data set would therefore increase the number of code-related revisits, since a typical refinding search could result in two code-related revisits, one for the search results page and one for the destination page. For this reason, we chose to filter visits to the Google search engine from our data set. However, no other search engines were filtered in this way. If participants used other search engines to find code-related web pages, the measured code-related recurrence rate will be somewhat higher than if all search engine result pages had been filtered.

Another possible threat is a potential bias in our page classification algorithm. Because this algorithm was tuned for high precision, it necessarily ignores a significant number of code-related pages. It is possible that the pages classified as code-related by our classifier also tend to be revisited more or less frequently than code-related pages in general. Addressing this

weakness would require a more sophisticated classification procedure, one which can achieve broader recall without sacrificing precision.

Chapter 4: Reverb

In this section, we discuss the design and implementation of Reverb, a tool which proactively recommends web pages from a developer's browsing history, based on the Java code currently being viewed inside the Eclipse IDE. We begin with a motivating example, to illustrate the types of coding tasks Reverb can facilitate.

4.1 Usage Scenarios

A developer, Helen, is writing a Java application which needs to retrieve content from various web sites. She chooses to use Apache's HttpClient library to establish connections with web servers and download pages. While developing her initial prototype, she discovers the detailed HttpClient tutorial on the Apache website.[4] Following the example in the tutorial, she uses the `DefaultHttpClient` class to make web requests in her prototype application.

A week later, Helen decides to parallelize her prototype so that multiple pages can be downloaded concurrently on separate threads. She opens the code for her `WebPageDownloader` class in Eclipse. The Reverb view instantly updates with a list of related links she had previously viewed, including several links to the HttpClient tutorial (Figure 4). She clicks on the second link to open up the chapter on connection management in her browser. Reviewing that page, she sees that she can configure the `DefaultHttpClient` with an instance of `PoolingClientConnectionManager` to ensure safe access from multiple threads.

Two months on, Helen is working on a new project where she must access a RESTful web service provided by another company. She decides she can do this quickly using the HttpClient library. She simply types `HttpClient` in to the Java code editor and clicks the refresh button inside the Reverb view. The view instantly updates with the links to the tutorial she used in the earlier project.

---

[4] See http://hc.apache.org/httpcomponents-client-ga/.

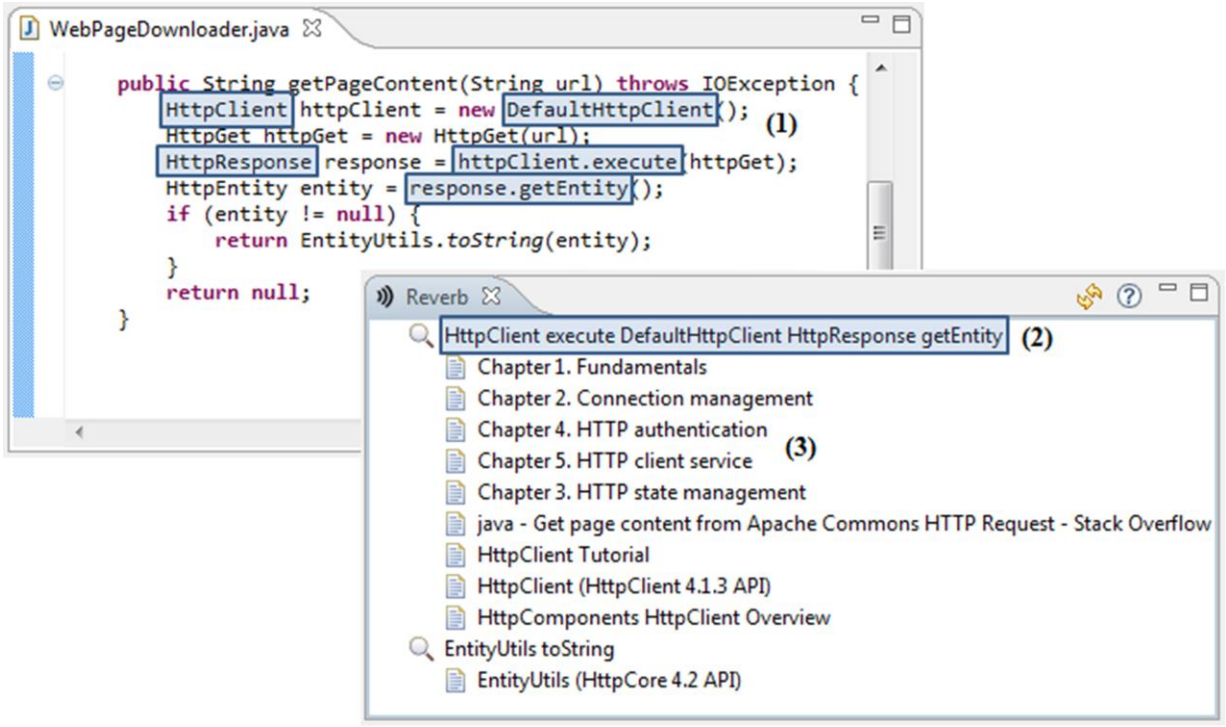

Figure 4. Reverb view inside Eclipse, showing query constructed from code elements in editor window. The code elements highlighted on the left (1) are used to build a query (2) against the developer's browsing history. Result links (3) are grouped according to the query they matched.

This example shows two key use scenarios for Reverb. The first is in viewing and modifying previously written code. Reverb can instantly surface links to pages that the developer explored when the code was originally written. These pages can remind the developer of the reasons underlying design and implementation decisions. These will also be pages that the developer may need to refer to if a bug needs to be fixed, or a new feature added.

The second scenario where Reverb may be useful is in writing new code which uses libraries the developer has worked with previously. Newly-entered code does not need to be saved or even compilable for Reverb to build queries from it. Often the name of a type is all that is needed to bring up useful references.

HttpClient was chosen as an example library not just because it has a detailed tutorial available on the web. It also depicts the case where two distinct versions of a library are in common use. Versions 3 and 4 of HttpClient are quite different, with many version 3 classes

deprecated in favor of alternate implementations in version 4. A Google search for HttpClient returns results for both versions. As of 6/1/2012, the first two results are for version 3, while the next two are for version 4. The developer must examine the result list carefully to determine which results apply to the version she wishes to use. In contrast, Reverb's ranking algorithm ensures that the pages most frequently and recently visited by the individual developer are at the top, so the desired version is immediately accessible.

While Reverb's current implementation works only for Java code, our approach is applicable to any statically-typed programming language. Aspects of our approach which depend on static typing are discussed in section 4.4.

4.2 Implementation

Figure 4 shows the user interface provided by Reverb, a view inside the Eclipse IDE. The view contains a list of page recommendations, grouped according to the code elements to which they relate, and ranked by content relevance, as well as the frequency and recency of visits to the page. Each time the developer clicks in a new location in the code editor, the view is updated. The heading for each group is a simplified version of the queries which returned these results. For each hit, only the page title is currently shown. Hovering over the hit shows a tooltip containing the page URL, and double-clicking on the hit opens the page in the browser.

Figure 5 shows Reverb's overall architecture. In addition to the Eclipse plugin, the tool includes a separate indexing service, as well as browser extensions for Chrome and Firefox. On a click event, the Eclipse plugin extracts the names of code elements, such as types referenced and methods declared or invoked, from the visible region of the current source file. These code element names are sent to the indexing service, which converts them in to queries. The queries are run against a full-text index of the developer's browsing history. The indexing service re-ranks the results returned by the indexing engine using frequency and recency information and sends them to the plugin.

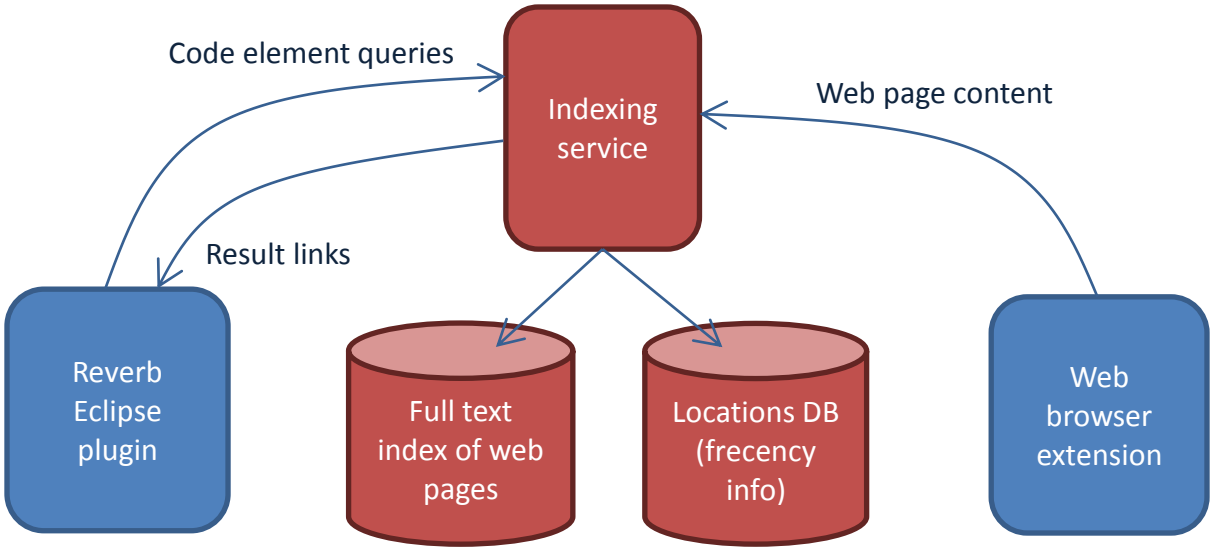

Figure 5. Reverb architecture.

The Chrome and Firefox browser extensions are responsible for sending page content to the indexing service whenever a new page is loaded. The following sections provide more detail on the components of the architecture, features of the design, and design alternatives that were considered. The source code for Reverb is available on Google Project Hosting.[5]

4.2.1 Browser Extensions

The Chrome and Firefox browser extensions are responsible for monitoring page loads and transferring the content of each page to the indexing service. On a page load event, a 5-second timer is started. When this timer fires, if the tab containing the page is still open and still displays the same location, then the page's HTML is extracted and sent to the indexing service. The timer serves three purposes. First, if the page was closed immediately by the user, it is likely of no interest and should not be added to the index. Second, the timer allows scripts on the page to finish loading dynamic content. Third, deferring the extraction and transfer of page content ensures that the browser extension has minimal impact on page load time.

---

[5] See http://code.google.com/p/reverb-bookmarks/.

A design consideration for the browser extension was whether all pages should be indexed, or if we should limit the index to pages classified as code-related. User concerns around sensitive content, page load times, and disk space usage argued for limiting the scope of pages indexed. However, in our formative study, we had seen the difficulties in accurately classifying pages. While the heuristics we used in the study had high precision, their recall was limited. Expanded recall would come at the cost of precision, meaning sensitive content might still find its way in to the index.

Through the 5-second timer and by delegating the page transfer to a separate thread, impact of the browser extension on page load times was minimized. Some testing and back-of-the-envelope calculations indicated that an index of pages visited over six months was unlikely to exceed 60 MB in size. Based on these considerations, we decided to proceed with indexing all pages, while providing an option for users to exclude selected domains from indexing.

To further reduce the performance impact of indexing all web pages, the browser extension maintains an in-memory history of the last 100 pages indexed, and ensures that a given page is only indexed once per day.

4.2.2 Indexing Web Pages

The indexing service is responsible for indexing page content sent by the browser extensions. Systems such as Strathcona [15] and Sourcerer [20] build indexes for source code repositories using structural information within the code. However, structural links are difficult to extract from source code embedded in web pages, because type references may not be resolvable. Consequently, Reverb's indexing approach is code-structure-unaware. It uses a general-purpose text indexing engine, Lucene,[6] to build the index of pages visited by the developer.

The Lucene records created by the service contain two fields: page title and page content. Preprocessing of the page prior to indexing is minimal. As in the tool for the formative study, the jsoup library is used for HTML processing. `<script>` elements are removed from the

[6] See http://lucene.apache.org.

page, and then the text content of the remaining HTML elements is concatenated and indexed. To reduce disk space requirements, only the inverted index is stored, not the full content of the pages.

The indexing service also maintains a SQLite database of locations visited, similar to the browsing history database maintained by Firefox and Chrome. 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. Details of the frecency calculation are discussed below, in section 4.2.5.2.

Lucene supports near real-time retrieval of newly-added content. So a page may be recommended in the IDE almost immediately after the developer has visited it for the first time.

4.2.3 Eclipse Plugin

The Eclipse plugin monitors mouse events in the Java code editor. The plugin updates the list of related links proactively: whenever the user 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 file that is visible and extracts the names of specific code elements.

Source code elements extracted currently include:

a) Type declarations
b) Type references (e.g., in field, variable, or parameter declarations)
c) Static field references (e.g., constants)
d) Method declarations which override a method in a parent class or implement a method from an interface
e) Method invocations

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.

Code does not have to compile successfully for code element information to be extracted. Here we rely on the ability of the Eclipse Java Development Tools (JDT) to generate a partial AST for code that does not yet compile. For code that does compile, the AST will contain the full type information for each code element, including the package name and type name. If code does not yet compile, the package name may not be present.

After extracting code elements from the AST, the Eclipse plugin translates them in to a language-independent representation and sends them to the indexing service. In this representation, each code element is described by a structure containing four fields:

a) Code element category (type declaration, type reference, static field reference, etc.)
b) Package name (if available in the AST)
c) Type name
d) Method or field name (if applicable for the code element category)

Code elements in a range of object-oriented programming languages can be described in this representation, allowing the indexing service to remain programming-language-agnostic. Only the code element extractor within the Eclipse plugin is aware of language-specific details.

4.2.4 Query Construction

The indexing service is responsible for constructing the Lucene query from the list of code elements provided by the Eclipse plugin. When the package name is provided, the service favors precision by including it in the query. Using the type name alone can result in false positives, because types in different API's (or even different programming languages) can have the same name. Adding the package name, when it is available in the AST, makes queries more selective.

Lucene query processing uses a vector space model to match queries with documents, but allows Boolean operators in the query to require specific terms to be present or absent.[7] When the package name 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`). An example of the Lucene query generated for calls to the `execute()` and `getParams()` methods of `HttpClient` is shown in Figure 6. In Lucene's query syntax, the `+` operator indicates an element that must be matched for a result to be returned.

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

Figure 6. Lucene query generated for calls to the `execute()` and `getParams()` methods of `HttpClient`.

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 enough on its own. The regular expression described in section 3.2 is used to decide whether a type name is a camelCase identifier, and therefore unlikely to match non-code-related terms.

As shown in Figure 6, multiple code elements associated with the same Java type (e.g., invocations of different methods belonging to the type) are grouped in to a single query. Multiple queries may be generated if the code elements extracted by the Eclipse plugin belong to multiple Java types. Our approach to combining and ranking the results for multiple queries is discussed in the next section.

[7] See http://lucene.apache.org/core/old_versioned_docs/versions/3_3_0/queryparsersyntax.html.

4.2.5 Ranking and Grouping of Results

At a high level, result pages 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; then they are grouped according to the types in the code that they match. This section describes our ranking and grouping approach in more detail.

4.2.5.1 Adjustments to Lucene Scoring Function

Given a query, Lucene assigns a relevance score to documents in its index according to how frequently the query terms appear in each document. More specifically, Lucene uses a vector space model with tf/idf normalization to assess the relevance of documents to the query.[8] A few customizations were made to the default Lucene scoring function based on Reverb's specific requirements.

- 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.
- 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. For our purposes, this was too strong a dependence. We forced the *coord* value to one, resulting in a dependence that was closer to linear.
- 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

[8] See http://lucene.apache.org/core/old_versioned_docs/versions/3_3_0/api/all/org/apache/lucene/search/Similarity.html.

was unhelpful, dramatically favoring shorter web pages over longer ones. As with the *coord* value, we forced the *lengthNorm* to one.

4.2.5.2 Reranking Results

After running code element queries against the Lucene index and retrieving the top 20 results for each query, the indexing service performs a few additional steps to determine the final list of 10 results which is returned to the Eclipse plugin.

First, it merges results from separate queries which correspond to the same web page. The Lucene relevance scores from the separate queries are added together. The goal is to ensure that web pages which match more elements in the current code context score higher than pages which match fewer. In the example shown in Figure 4, we would like to ensure that web pages which mention both `HttpClient` and `HttpResponse` are ranked more highly than pages that mention either element on its own.

Next, the indexing service 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.[9] 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 [9]. 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:

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

An advantage of an exponential decay function is that only the last visit time needs to be stored. The current frecency score can be calculated by multiplying the frecency score at the last visit time by the decay factor for the time difference.

[9] See https://developer.mozilla.org/en/ The_Places_frecency_algorithm.

The time constant τ was chosen to ensure a fairly long half-life of 6 months, weighting frequency more heavily than recency. For scoring purposes, two visits six months ago are worth one today. This half-life is relatively long in light of the revisit distribution shown in Figure 1. We wanted 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 tuning of τ further in the evaluation section, below.

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.

4.2.5.3 Grouping Results

In an early prototype of Reverb, results were presented in a flat list, ordered simply by their overall score. Based on our own use of the tool, we quickly realized that this presentation was problematic. Page title alone was often not sufficient to remind us of the topic of a page. Compounding the problem, results relating to different code elements in the editor window were intermingled with each other.

Our solution to this problem was to group results together, according to the queries they matched. As discussed in section 4.2.3, each query corresponds to a single Java type, so

grouping by query will pull together results that match the same Java type. The grouping algorithm starts with the top 10 results from the ranking procedure described earlier and proceeds 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. Queries are then ranked according to their highest scoring result page.
3. Finally, query merging occurs. 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 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 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 hits discuss.

4.2.5.4 Filtering Duplicates

Handling pages with distinct URL's but near-identical content is an often-cited challenge for search engines. 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 URL's 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 URL's 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.

4.2.6 Privacy Protection

Users' browsing histories contain sensitive information. Reverb's browsing history data is stored in access-controlled files on the user's hard drive. The Eclipse plugin and browser extensions communicate through operating system pipes. These are protected with an access control list (ACL) which grants read/write permissions only to processes running under the same user ID.

In addition, in the settings for the browser extension, the user can configure domains which should never be indexed, either for security reasons (e.g., their banking web site) or to avoid unnecessary disk space consumption (e.g., the Google search domain).

4.3 Design Alternatives

In this section we discuss some key decisions we made in designing Reverb, and alternatives we considered.

Rather than requiring the developer to select a particular code element or region of code for which queries should be generated, Reverb proactively initiates queries for the code currently visible in the editor. The benefit of a proactive approach is the minimization of developer effort to locate pages. The developer can quickly scan the list to see if the page of interest is there. Clicking on a link opens it immediately in the browser. Reducing the effort to this level means that developers may choose to revisit pages even in cases where, without Reverb, they would rely on memory. This can yield substantial benefits by allowing developers to quickly home in on the correct solution and avoid dead ends.

Reverb builds queries out of the names of code elements. A simpler approach to query construction is to use the source code text itself as the query. The search engine then determines the documents in the index that are closest to the query according to some

measure, such as cosine similarity. However, in the study of Bacchelli et al. [3], this approach was found to yield poor precision in linking source code with email messages. In that study, the best combination of precision and recall was achieved with a technique similar to ours, relying on the names of code elements (specifically Java class names) to identify matches.

In building queries for the indexing service, the Reverb plugin looks only at the code that is currently visible in the Java code editor. An advantage of this strategy is that the developer can quickly understand why a link is being recommended. A drawback is that the list of results changes frequently. This may be distracting for developers. In addition to reducing distractions, expanding the scope of context considered could have a benefit in result relevance. For example, if the developer examines three methods over the course of five minutes, and all three methods reference a particular class, it may make sense to boost the scores of web pages that mention that class. Or if a web page receives a particularly high score based on the method currently being edited, it may be better to retain that result in the Reverb view, even when the developer briefly navigates to a different method. In our evaluation of the tool, we explore this question further.

4.4 Limitations of the Tool

There are a number of clear limitations in the current implementation of Reverb. First, even when combined with the query for the result group, the page title is sometimes not sufficient to remind the user of the topic of a page. There are a couple of ways that this problem might be addressed. Snippets of page content could be embedded inside the Reverb results list. Another possibility would be to display a thumbnail snapshot of the page itself, similar to Chrome's New Tab page, which shows snapshots of the user's most frequently visited destinations.

A limitation of our overall approach lies in the use of package names to improve query precision. Using package names in this way means that our technique is mainly applicable to statically-typed programming languages. While dynamically-typed languages often include a module feature similar to Java's packages, module information is not usually available within

the AST of a dynamically-typed program. The absence of module information would make it more difficult to construct precise queries.

A third limitation relates to common code elements which show up frequently both in code and code-related web pages. In Java, the core data types, such as `String` and `Integer`, the collections classes and interfaces, such as `List`, and other frequently-referenced classes, such as `IOException`, fall in to this category. These are elements for which the developer is unlikely to require page recommendations, because their usage is typically well understood. In addition, code-related web pages will often match queries based on one of these common elements, even though the primary focus of the page is not the code element in question. This problem occurs in general information retrieval as well, and tf/idf normalization is intended to mitigate it. However, in the context of the developer's overall browsing history, terms like `String`, `Integer`, and `IOException` are still relatively uncommon, and the impact of the inverse document frequency normalization on the scores for these matches is small.

Our current approach to this problem is two-fold. First, in the Eclipse plugin's code element extractor, similar to the list of stop-words used by some text indexers, we define a list of "stop-types": types which are never included in the code element lists sent to the indexing service. The core Java data types are included in this list.

Second, we allow the user to manually extend this list of stop-types. When the user right-clicks on the header for a result group, she is given the option to block all results for the first type in the query. If she chooses to block the type, the preference is stored by the indexing service, and no further queries are issued for that type.

Reverb requires version 1.6 or later of Java and Eclipse 3.7 or later. Also, Reverb is currently limited to running on the Windows operating system (Vista or later). This is because Reverb uses native OS components (named pipes) for interprocess communication (IPC), to ensure that access to the indexing service can be protected through ACLs. Similar IPC mechanisms exist on Linux and Mac, and work is in progress to make Reverb available on these platforms.

Chapter 5: Evaluation of Reverb

In this section, we present results from a small field study of Reverb. In this study, we were interested in measuring how often users clicked on Reverb's recommendations, and gathering qualitative feedback on the quality of the recommendations, the usefulness of the tool, and areas for improvement.

In addition to recording recommendations clicked, we also tracked all pages participants visited in the browser. This allowed us to detect "blind" hits: code-related revisits that were not initiated through Reverb, but which Reverb successfully predicted. Participants will likely click on one or two recommendations just to familiarize themselves with how a tool works. Revisits initiated outside of the tool are interesting, because they are not vulnerable to this novelty effect. If the tool predicts such revisits, it gives a strong confirmation that the tool is providing useful recommendations. A field study was chosen over a lab-based study, as it would allow more time for code-related revisits of both types to occur.

5.1 Study Setup

For purposes of the study we made changes to the indexing service and the Eclipse plugin. The indexing service was modified to log the recommendations that it generated, along with the pages that were actually visited in the browser. Pages visited were classified as code-related or non-code-related, using the heuristics described in section 3.2. As in the formative study, page locations were anonymized in the logs, with the URL replaced by an integer identifier.

The plugin was configured so that even when the Reverb view was not open inside of Eclipse, code element lists continued to be sent to the indexing service and recommendations continued to be generated and logged. This allowed us to detect blind hits even when the Reverb view was closed.

User actions within the Eclipse plugin were also logged. Events were logged each time the user clicked on a recommendation, opened or closed the Reverb view, removed a page from the index or blocked a Java type from being included in queries.

In choosing a duration for the field study, we sought to balance the need to collect sufficient data with the goal of minimizing participants' time commitment. In the formative study, we had found many one-hour periods where participants initiated at least one code-related revisit. In periods of active coding activity, we expected to see one or more code-related revisits per hour. With at least five such revisits, assuming a prediction accuracy of 20% or greater, we expected to see some successful predictions. So we estimated that five to six hours would be just enough to begin to assess the tool's effectiveness. Accordingly, we set the study duration at six hours of active coding activity.

To detect periods of active coding activity, we modified the monitor in the Eclipse plugin to track all mouse and keyboard events in the Java code editor. The activity monitor divided time in to 15-minute intervals. If an interval included at least one interaction in the Java code editor, it was counted towards the six hour quota. In the study, it took participants between one and seven days to accumulate six hours of coding activity.

Half-way through the study and at the end of the study, participants were prompted to rate and comment on each recommendation on which they had clicked. At the end of the study period, participants were asked to complete a survey about their impressions of the tool and ways it could be improved.

In order to allow participants to begin receiving useful recommendations right away, they were given the opportunity to prepopulate their browsing history index. At the end of installation, a dialog was displayed, asking participants whether they would like to index their Chrome and Firefox browsing history from the past three months. Participants were encouraged to perform this step and all did so. The mechanics of this indexing step were similar to the procedure followed by the page classification tool in the formative study (section 3.1). The content of each page in the browser's history database was downloaded and added to the index. The limitations described in section 3.5 around dynamic content, cookies, and access-controlled pages also apply here.

We advertised the study on Eclipse mailing lists, through contacts at various companies, and within the UBC computer science department. Participation was completely anonymous:

developers could download the tool and participate in the study without interacting with us in any way. Three participants completed the study.

5.2 Results

Table 2 summarizes the results of the field study. The *S@10* metric describes the percentage of code-related revisits that were successfully predicted by Reverb. A revisit is considered to be successfully predicted when it was one of the top 10 results suggested by Reverb *at the time the revisit occured*. When all code-related revisits are included, the average S@10 value across the three participants was 35%. As mentioned earlier, we were also interested in the blind hit rate: the hit rate calculated when revisits initiated within the tool are excluded. The blind S@10 value was 18%.

Table 2. Summary of results from Reverb field study.

Participant	1	2	3	Mean
View open %	78%	35%	89%	67%
Recommendations clicked	2	2	3	2.3
Average rating (/5)	2.5	3.5	n/a	3
S@10, all revisits	25%	57%	22%	35%
S@10, blind revisits	0%	40%	13%	18%

Table 2 also provides general information on participants' use of Reverb. *View open %* represents the fraction of active development time during which the Reverb view was displayed in Eclipse. It is measured as the ratio of the 15-minute intervals where the view was open to the total intervals in which activity was detected in the Java code editor. Two of the three participants kept the Reverb view open for most of the time they were working in the IDE.

Recommendations clicked shows the number of Reverb recommendations the participant clicked on during the study. *Average rating* is the mean rating that the participant assigned to the recommendations on which they clicked. The rating scale was from 1 to 5, with 1 labeled as "not useful" and 5 labeled "very useful". Participants clicked on two to three recommendations each. On average, participants rated the links they clicked as moderately useful (3 out of 5).

Two of the three participants (1 and 2) completed the end-of-study survey. Participant 2's feedback was very enthusiastic, saying "Yes, it provided recommendations which are very useful to me," and indicating they intended to keep the view open in Eclipse on an ongoing basis. Participant 1 stated that they were doing exploratory testing and did not use Reverb, but would make use of the tool if they were working with old code. Both participants 1 and 2 mentioned difficulties in identifying links from the page title alone. Participant 2 suggested that last access time and visit count would be helpful additions to the Reverb view to remind them of page content.

Table 3 shows detailed information on code-related visits, revisits, and Reverb recommendations. This data presents two sets of values, one where revisits initiated through Reverb are included, and one where these revisits are excluded. The meanings of *code-related visits*, *code-related revisits*, and *code-related recurrence rate* are the same as discussed in the formative study. As before, a 15-minute filter was used to qualify these revisits. For this small sample, the code-related recurrence rate is slightly higher than was measured in the earlier study (20% including Reverb-initiated revisits and 16% excluding these revisits, as compared with 13.7% in the formative study).

In some cases, Reverb recommended pages that were not classified as code-related. In classifying pages, the indexing service used the same heuristics as in the formative study. These heuristics are optimized for precision rather than recall. Based on our own experience of using the tool, it is very rare for Reverb to recommend a page which is completely unrelated to development. The cases where Reverb recommended non-code-related pages likely reflect the limited recall of our classifier, rather than false positives. In Table 3, the *revisits recommended* value shows the total number of revisits (code-related and non-code-related) that Reverb recommended at any point prior to the revisit occurring. The *CR revisits recommended* shows the corresponding value when only pages classified as code-related are included.

Table 3. Revisit and recommendation data from Reverb field study.

	Including Reverb-initiated revisits.				Excluding Reverb-initiated revisits.			
Participant	1	2	3	Mean	1	2	3	Mean
Code-related visits	37	28	38	34	36	26	37	33
Code-related revisits	4	7	9	6.7	3	5	8	5.3
Code-related recur. rate	11%	25%	24%	20%	8%	18%	22%	16%
Revisits recommended	4	6	5	5	2	4	2	2.7
CR revisits recommended	2	6	2	3.3	1	4	1	2
S@10, initial settings	25%	57%	22%	35%	0%	40%	13%	18%
S@10, optimized	67%	80%	25%	57%	33%	60%	13%	35%

As opposed to the number of code-related revisits recommended, the S@10 metric shows the percentage of code-related revisits which were successfully *predicted* by Reverb. As described earlier, a revisit is considered to be successfully predicted when it is one of the top 10 results recommended at the time the revisit occurs. At the start of this section, we stated the S@10 values achieved with the Reverb configuration that was provided to study participants. These values are shown in Table 3 in the row labeled *S@10, initial settings*. In the next section, we discuss how parameters were tuned to obtain the values shown for *S@10, optimized*.

5.2.1 Parameter tuning

As discussed in section 4.3, we were interested to see if the quality of recommendations could be improved by considering a larger scope of code context information. In the configuration provided to participants, Reverb generated a new set of recommendations each time a click event occurred, provided the visible region in the code editor was different from the previous click. We say this implementation has a recommendation window of 10. At this window size, the mean S@10 value was 35%, reaching a high of 57% for one participant.

Since we logged all recommendations that Reverb generated, we could determine the hit rate that would have been achieved by gathering recommendations over the previous two visible regions, or the previous three. In this simulation, for a recommendation window of 20, we collected the previous 20 recommendations generated by the tool, and ranked them using the scoring scheme already discussed. We then determined the percentage of revisits which occurred in the top 10 with this new window. For a window of 30, we used the previous 30 recommendations generated.

In the same simulation, we wanted to tune the ranking parameters τ and α, discussed in section 4.2.5.2. τ is the constant used in calculating the decay of the frecency boost attached to a previous visit. α is the geometric weighting of relevance relative to frecency in the ranking function. We sought to maximize the S@10 metric while varying the recommendation window, the decay constant τ, and the weighting factor α.

With the small sample from the field study, the best S@10 value 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 α weighting parameter remained 1. With these settings, the average hit rate was 57%, reaching a high of 80% for one participant and a low of 25% for another. The 57% hit rate is promising, suggesting that local code context can indeed help in predicting code-related page revisits. For comparison, we can consider the hit rate that would result if 10 previously-visited code-related pages were chosen at random. For the field study, we do not know the number of code-related pages in each participant's locations database. However, from the formative study, we know that, on average, participants accumulated about 190 distinct code-related locations over three months. The expected hit rate for a choosing 10 at random would be 10/190 = 5%.

It is important to note that the hit rate in the simulation 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.

It is worth highlighting that changes in the list of recommendations presented to users could affect their behavior. The 57% 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 they did in the original study. For this reason, in Table 3 we also report 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 S@10 value of 35%.

5.3 Discussion and Future Work

To get the most benefit from Reverb, changes in developer behavior are needed. When about to modify existing code, rather than immediately running a search in the browser, the Reverb-oriented developer would first open the code about to be modified, and check if the pages needed are recommended by the tool. Changes to ingrained behaviors take time. Six

hours of coding activity is probably not long enough for participants to start using Reverb in these ways. The current field study provides some validation that the tool can provide useful recommendations. A longer-term study, to assess whether developers make more use of Reverb over time, remains as future work.

The S@10 metric used to evaluate the tool relates to recall: the percentage of code-related revisits which Reverb successfully predicted. Not explored in our study is the precision of Reverb's results. One precision metric would be the percentage of Reverb recommendations which were actually visited by the user. For a contextual recommender, which generates suggestions even when the developer is engaged in activities for which they need no assistance, this metric will be very low. A more appropriate measure might be the percentage of recommendations that a user judges to be relevant to the code under development and *potentially* of use. We believe that one of Reverb's strengths is the user-perceived relevance of results, arising from the combination of highly selective queries with a ranking system that favors frequently- and recently-visited pages. Also enhancing users' perception of relevance is the fact that the connection between the local code context and page recommendations is usually clear. In the field study, developers were asked to rate only those links that they clicked on. It would be interesting to conduct a study in which participants rate a large number of links, allowing us to measure the overall percentage of recommendations which are deemed relevant to the code. This study is also left for future work.

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, there are a couple of reasons to consider it. If the gravity towards familiar, browser-based methods of refinding is very strong, putting recommendations in the browser may increase their likelihood of being used. Or if users tend to alternate code-related refinding with searches for new content, that would also argue for making Reverb's bookmarks available in the browser.

Another possibility for future work relates to the nature of the context on which Reverb relies in making recommendations. Currently, Reverb looks for similarities in the content of code

and pages visited. An alternative is to look for 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 a 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.

Chapter 6: Conclusion

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% (σ = 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 have built Reverb. Reverb proactively recommends previously-visited web pages, based on the code currently visible in the IDE. 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 developer's browsing history. It combines cosine similarity with frequency and recency of page visits to determine the rank of recommendations, and aggregrates results according to the code element queries they match.

We have conducted a small field study of Reverb. Our results suggest that local code context is indeed predictive of web page revisits, with 57% of code-related revisits being predicted by an optimized version of Reverb's ranking algorithm.

As outlined in section 5.3, there are a number of possible areas for future work.

- A longer-term field study would help in assessing whether users prefer Reverb's revisitation support to more familiar techniques for refinding web pages.
- We hypothesize that even when developers do not choose to click on any of Reverb's recommendations, they still perceive many of them to be relevant to their code and potentially useful. A user study is needed to validate this hypothesis.
- Content snippets and thumbnail images could be explored as mnemonic cues to remind users of the topic of a page.

- Users will tend to gravitate towards familiar, browser-based methods of refinding web pages. As an alternative to displaying recommendations in the IDE, approaches to making Reverb's recommendations available in the browser could be investigated.
- Reverb's recommendations currently rely on content similarity between code and web pages. It would be interesting to explore whether temporal associations between code interactions and web page visits could be used to improve Reverb's recommendations.

References

1. Adar, E., Teevan, J., and Dumais, S.T. Large scale analysis of web revisitation patterns. *Proceedings of the twenty-sixth annual SIGCHI conference on Human factors in computing systems*, ACM (2008), 1197–1206.

2. Alnusair, A., Zhao, T., and Bodden, E. Effective API navigation and reuse. *2010 IEEE International Conference on Information Reuse and Integration*, IEEE (2010), 7 –12.

3. Bacchelli, A., Lanza, M., and Robbes, R. Linking e-mails and source code artifacts. *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, ACM (2010), 375–384.

4. Billsus, D.-A., Hilbert, D.M., Trevor, J.J., Culy, C.D., Denoue, L., and Golovchinsky, G. Indexing for contextual revisitation and digest generation. U.S. Patent 7,363,294, April 22, 2008.

5. Brandt, J., Dontcheva, M., Weskamp, M., and Klemmer, S.R. Example-centric programming: integrating web search into the development environment. *Proceedings of the 28th international conference on Human factors in computing systems*, ACM (2010), 513–522.

6. Brandt, J., Guo, P.J., Lewenstein, J., Dontcheva, M., and Klemmer, S.R. Two studies of opportunistic programming: interleaving web foraging, learning, and writing code. *Proceedings of the 27th international conference on Human factors in computing systems*, ACM (2009), 1589–1598.

7. Brank, J., Frayling, N.M., Frayling, A., and Smyth, G. Predictive Algorithms for Browser Support of Habitual User Activities on the Web. *Proceedings of the 2005 IEEE/WIC/ACM International Conference on Web Intelligence*, IEEE Computer Society (2005), 629–635.

8. Cockburn, A., Greenberg, S., and Greenberg, S. Getting Back to Back: Alternate Behaviors for a Web Browser's Back Button. *Proceedings of the 5th annual Human Factors and the Web Conference*, (1999).

9. Cormode, G., Shkapenyuk, V., Srivastava, D., and Xu, B. Forward Decay: A Practical Time Decay Model for Streaming Systems. *IEEE 25th International Conference on Data Engineering*, IEEE (2009), 138 –149.

10. Goldman, M. and Miller, R.C. Codetrail: Connecting source code and web resources. *Journal of Visual Languages & Computing 20*, 4 (2009), 223–235.

11. Gray, W.D. and Boehm-Davis, D.A. Milliseconds matter: an introduction to microstrategies and to their use in describing and predicting interactive behavior. *Applied Journal of Experimental Psychology 6*, 4 (2000), 322–335.

12. Hailpern, J., Jitkoff, N., Warr, A., Karahalios, K., Sesek, R., and Shkrob, N. YouPivot: improving recall with contextual search. *Proceedings of the 2011 annual conference on Human factors in computing systems*, ACM (2011), 1521–1530.

13. Hartmann, B., Dhillon, M., and Chan, M.K. HyperSource: bridging the gap between source and code-related web sites. *Proceedings of the 2011 annual conference on Human factors in computing systems*, ACM (2011), 2207–2210.

14. Hoffmann, R., Fogarty, J., and Weld, D.S. Assieme: finding and leveraging implicit references in a web search interface for programmers. *Proceedings of the 20th annual ACM symposium on User interface software and technology*, ACM (2007), 13–22.

15. Holmes, R. and Murphy, G.C. Using structural context to recommend source code examples. *Proceedings of the 27th international conference on Software engineering*, ACM (2005), 117–125.

16. Kaasten, S. and Greenberg, S. Integrating back, history and bookmarks in web browsers. *Proceedings of the SIGCHI conference on Human factors in computing systems*, ACM (2001), 379–380.

17. Kawase, R., Papadakis, G., Herder, E., and Nejdl, W. Beyond the usual suspects: context-aware revisitation support. *Proceedings of the 22nd ACM conference on Hypertext and hypermedia*, ACM (2011), 27–36.

18. LeeTiernan, S., Farnham, S., and Cheng, L. Two methods for auto-organizing personal web history. *Proceedings of the SIGCHI conference on Human factors in computing systems*, ACM (2003), 814–815.

19. Lieberman, H., Fry, C., and Weitzman, L. Exploring the Web with reconnaissance agents. *Commun. ACM 44*, 8 (2001), 69–75.

20. Linstead, E., Bajracharya, S., Ngo, T., Rigor, P., Lopes, C., and Baldi, P. Sourcerer: mining and searching internet-scale software repositories. *Data Min. Knowl. Discov. 18*, 2 (2009), 300–336.

21. Mandelin, D., Xu, L., Bodík, R., and Kimelman, D. Jungloid mining: helping to navigate the API jungle. *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, ACM (2005), 48–61.

22. McKenzie, B. and Cockburn, A. An Empirical Analysis of Web Page Revisitation. *Proceedings of the 34th Annual Hawaii International Conference on System Sciences*, IEEE Computer Society (2001), 5019–.

23. Milic-Frayling, N., Jones, R., Rodden, K., Smyth, G., Blackwell, A., and Sommerer, R. Smartback: supporting users in back navigation. *Proceedings of the 13th international conference on World Wide Web*, ACM (2004), 63–71.

24. Milic-Frayling, N., Sommerer, R., and Rodden, K. WebScout: support for revisitation of Web pages within a navigation session. *Proceedings of the IEEE/WIC International Conference on Web Intelligence*, IEEE (2003), 689 – 693.

25. Obendorf, H., Weinreich, H., Herder, E., and Mayer, M. Web page revisitation revisited: implications of a long-term click-stream study of browser usage. *Proceedings of the SIGCHI conference on Human factors in computing systems*, ACM (2007), 597–606.

26. Rhodes, B.J. and Maes, P. Just-in-time information retrieval agents. *IBM Syst. J. 39*, 3-4 (2000), 685–704.

27. Sahavechaphan, N. and Claypool, K. XSnippet: mining For sample code. *SIGPLAN Not. 41*, 10 (2006), 413–430.

28. Sawadsky, N. and Murphy, G.C. Fishtail: from task context to source code examples. *Proceedings of the 1st Workshop on Developing Tools as Plug-ins*, ACM (2011), 48–51.

29. Stylos, J. and Myers, B.A. Mica: A Web-Search Tool for Finding API Components and Examples. *IEEE Symposium on Visual Languages and Human-Centric Computing*, IEEE (2006), 195–202.

30. Takano, H. and Winograd, T. Dynamic bookmarks for the WWW. *Proceedings of the ninth ACM conference on Hypertext and hypermedia*, ACM (1998), 297–298.

31. Tauscher, L. and Greenberg, S. How People revisit Web Pages: empirical findings and implications for the design of history systems. *International Journal of Human Computer Studies 47*, (1997), 97–137.

32. Ye, Y. and Fischer, G. Supporting reuse by delivering task-relevant and personalized information. *Proceedings of the 24rd International Conference on Software Engineering*, (2002), 513 –523.