HyperSource: Bridging the Gap Between Source and Code-Related Web Sites

Björn Hartmann, Mark Dhillon, Matthew K. Chan
Computer Science Division
University of California, Berkeley, CA 94720
bjoern@cs.berkeley.edu, {mark.dhillon, mattkc7}@gmail.com

ABSTRACT

Programmers frequently use the Web while writing code: they search for libraries, code examples, tutorials, and documentation. This link between code and visited Web pages remains implicit today. Connecting source code and browsing histories might help programmers maintain context, reduce the cost of Web page re-retrieval, and enhance understanding when code is shared. This note introduces HyperSource, an IDE augmentation that associates browsing histories with source code edits. HyperSource comprises a browser extension that logs visited pages; an IDE that tracks user activity and maps pages to code edits; a source document model that tracks visited pages at a character level; and a user interface that enables interaction with these histories. We discuss relevance heuristics and privacy issues inherent in this approach. Informal log analyses and user feedback suggest that our annotation model is promising for code editing and might also apply to other document authoring tasks after refinement.

ACM Classification: H.5.2 [Information Interfaces and Presentation]: User Interfaces – GUIs.

General terms: Design, Documentation

Keywords: code editors, browsing history, edit wear

INTRODUCTION

When programming, developers spend a significant amount of time outside their Integrated Development Environment (IDE) and inside their Web browser. In a lab study by Brandt, programmers spent 19% of their time on the Web [3]; in Goldman's field study, 23% of Web pages visited in temporal proximity to code edits were development-related [4]. Work-related uses of the Web include finding libraries and source code examples, consulting online documentation, forums, and question-and-answer sites, and managing project-related communication in software forges.

Despite this strong connection between Web browsing activity and code production, today's IDEs and browsers remain largely unaware of each other. We hypothesize that *making the connection between source code and visited pages explicit can benefit programmers* in multiple ways. First, for maintenance of one's own code, access to previously consulted resources can help re-establish context and shorten the time spent re-retrieving previously used resources (a common pattern in Web browsing [10]). Second, for code shared between developers, relevant Web resources can help project newcomers understand design decisions and become productive faster.

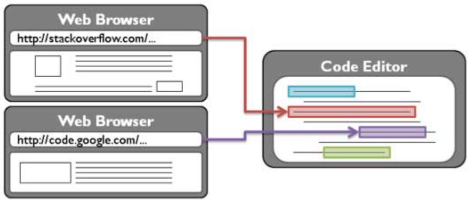
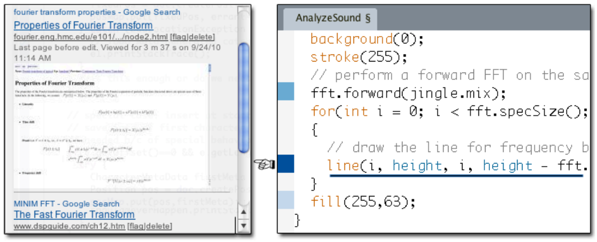


Figure 1: HyperSource associates Web pages visited by programmers with subsequent code edits.


fourier transform properties - Google Search
Properties of Fourier Transform
fourier.eng.hmc.edu/e101/.../node2.html [flag|delete]
Last page before edit. Viewed for 3 m 37 s on 9/24/10 11:14 AM

Properties of Fourier Transform

MINIM FFT - Google Search
The Fast Fourier Transform
www.dspguide.com/ch12.htm [flag|delete]

AnalyzeSound §

```
  background(0);
  stroke(255);
  // perform a forward FFT on the sa
  fft.forward(jingle.mix);
  for(int i = 0; i < fft.specSize();
  {
    // draw the line for frequency b
    line(i, height, i, height - fft.
  }
  fill(255,63);
}
```

Figure 2: Web history (L) associated with the current line of code (R). Gutter highlights provide scent.

To investigate these potential benefits, we developed *HyperSource*, an augmented IDE that associates browsing histories with source code edits (Figure 1). Conceptually, HyperSource builds upon *read* and *edit wear* [5] by collecting and showing sets of Web pages that were read while code was edited. HyperSource enables this association via a source document model that attaches metadata to text at the character level. Annotations are maintained through subsequent document edits (insertions, deletions, copy, cut, paste, undo and redo). As programmers navigate their source, a panel inside the IDE displays associated Web histories and enables interaction with them (Figure 2). HyperSource builds on previous work in aggregating relevant information artifacts for software developers (Mylar [6]) and on tools that integrate web information sources into the IDE for code documentation (CodeTrail [4]) and code examples (BluePrint [2]).

The main contribution over prior work is *an architecture that implicitly collects visited pages and associates them with lines of code that developers subsequently write*.

SHARING ANNOTATED CODE WITH HYPERSOURCE

Joe, a Computer Science student, is writing a visualization of Web server logs for his student organization. As he writes his code, he discovers a bug in a drawing library. An online forum post suggests a counterintuitive workaround. As he consults these Web pages, their URLs are tracked inside his IDE and attached to subsequent code edits.

The following semester, Jan, another member of the club, has been tasked with changing Joe's code to add richer visualizations. She is initially confused by the drawing code since it does not match the library's official documentation. Because Joe wrote the code with HyperSource, Jan can select the confusing statement and see links to the original forum posts Joe consulted. She thus uncovers the rationale for Joe's code and can complete her task.

HYPERSOURCE ARCHITECTURE

Tracking and visualizing visited pages requires four components: a browser extension that captures visited Web pages; an augmented editor that can manage captured pages; a source document data structure that maintains associations between code and web pages; and a user interface that enables interaction with this history.

Browser Extension Captures Pages

Whenever a user loads a new page in her web browser, a HyperSource browser extension collects the URL, title, and a thumbnail image of the rendered page. This page record is then sent to the IDE using inter-process communication. User interactions with the page in the browser may indicate that a page is especially relevant: our extension intercepts *copy events* that place page content on the system clipboard and marks corresponding pages as *copy sources*. In addition, the extension adds a button to the browser UI which sets an *explicit bookmark* flag.

Our prototype is written as an extension to the Google Chrome browser in JavaScript and uses XML-HTTP requests to communicate with the IDE. When the IDE is not running, no information is recorded.

IDE Stores and Associates Visited Pages With Code

The HyperSource IDE receives page visit data from the web browser and attaches it to the currently open document. We have implemented HyperSource for two open source IDEs: *Processing* [1], an editor for a Java dialect focused on multimedia programming; and *jEdit*, a universal text editor with support for multiple languages. In each case, the IDE runs an additional server thread that services requests from the browser extension.

What rules should guide the association of pages to code? HyperSource models user behavior as a cycle of alternating edit and browse phases: code is annotated with the set of pages viewed just prior to the current set of edits (Figure 3). This model is operationalized using a finite state machine (Figure 4): during browsing, visited pages are added to a queue; when the user edits code, all pages in the queue are assigned to new characters. When the user switches back to browsing, the queue is cleared, then filled again. This model does not decide whether browsing is code-related or not; that decision is based on heuristics described below.

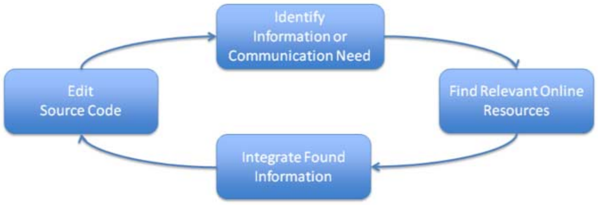

Identify Information or Communication Need
Find Relevant Online Resources
Integrate Found Information
Edit Source Code

Figure 3: The underlying model of user interaction.

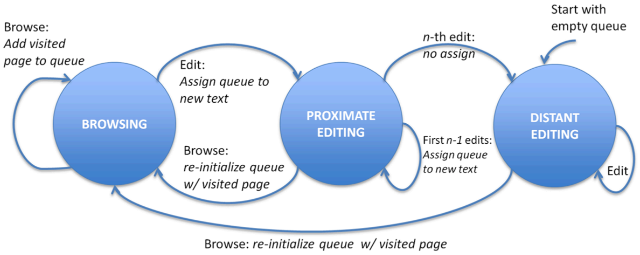

Browse: Add visited page to queue
BROWSING
Edit: Assign queue to new text
Browse: re-initialize queue w/ visited page
PROXIMATE EDITING
n-th edit: no assign
First n-1 edits: Assign queue to new text
Start with empty queue
DISTANT EDITING
Edit
Browse: re-initialize queue w/ visited page

Figure 4: State machine that implements the model.

Early prototype testing suggested that *edit locality* matters: not all edits that follow browsing are related to collected pages. To model locality, HyperSource assumes that edits that occur in temporal proximity to browsing are the most relevant; therefore, only the first *n* edits are annotated with visited pages. Based on pilot studies, we currently use *n*=2, ignoring whitespace. This is a conservative choice: users may have to search more, but are less likely to see annotations in irrelevant locations.

Source Document Tracks Associations Over Time

To maintain the link between source code sections and visited pages, HyperSource tracks associations at the character level. After Web pages are assigned to code, subsequent code edits may insert or remove text, duplicate text through copy and paste, and reverse previous operations through undo and redo. HyperSource maintains associations between characters and pages with *position tracking* (using *Java Swing Positions*): each character is identified by a position object that describes its location throughout the document's lifecycle. While the absolute offset of a character changes, its position object remains the same. Position objects have to be updated after every edit (Figure 5).

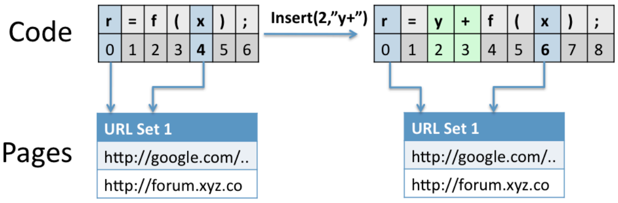


Figure 5: Character offsets change across edits; position tracking maintains associations.

HyperSource writes a shadow file to disk for every regular source file. It shares name and directory with the source file, but has a separate extension (`.hypersrc`). Both files are written during a save operation. Use of the shadow file is optional: collaborators can still read the original source file. If collaborators edit that source file without HyperSource, shadow and source file lose sync. Differencing algorithms could be used to re-sync the shadow file.

History View Provides Scent and Details

The user interface to HyperSource provides *information scent* [8] about which code sections have associated pages; and it provides methods to filter, view, and manage pages. Analogous to the way many IDEs indicate breakpoints, Information about which text contains linked pages is shown on a line-level in a sidebar (Figure 2). The number of items for a line is encoded in highlight saturation and brightness. Clicking on any line sidebar brings up a detailed view of the relevant web history. This view enables users to display complete or filtered results, and toggle between display styles to show page thumbnails, or increase display density with text-only links. In addition to viewing line items, user can also view all items in the text span of the current selection. When viewing text selections, the view panel updates automatically as the document cursor moves through the source document. While HyperSource associates pages to individual characters, the user sees annotations for lines or active selections. The granularity of tracking annotations is fine; the granularity of interacting with annotations is under user control.

FILTERING & PRIVACY

HyperSource addresses the shortcoming that today's tools do not capture the semantic connection between browsing and coding. However, care has to be taken that automatic, implicit capture does not lead to an inverse problem of capturing too many, potentially unwanted, links. Not all visited Web pages should be logged and displayed, for two main reasons: some pages might not be relevant to the programming project (they represent noise, not signal); and sharing of logged pages might expose confidential or private information to others. We discuss possible approaches to guard against these challenges.

Maximizing signal-to-noise ratio can be approached through techniques for estimating the relevance of individual pages and then tailoring the display of logged pages to emphasize important pages and de-emphasize or hide unimportant pages. HyperSource currently uses a set of heuristics based on user actions to coarsely segment pages into two classes. Pages are deemed relevant if:

❶ text was copied from the page into the source document;
❷ the developer marked the page in the browser extension;
❸ the page was the final page visited before editing.

To address privacy and confidentiality concerns, certain pages should not be logged in the first place. To prune the set of logged pages, HyperSource offers three filtering mechanisms: a blacklist of domain names; a "clear buffer" action that empties the internal queue; and widgets for individual log item removal. An additional approach, reserved for future work, would be to attempt to automatically classify pages as programming-related, irrelevant, or private through analysis of page content features.

USER EXPERIENCES WITH HYPERSOURCE

We conducted 3 informal procedures to understand benefits and challenges of the HyperSource annotation mechanism.

Data Collection Through a Programming Competition

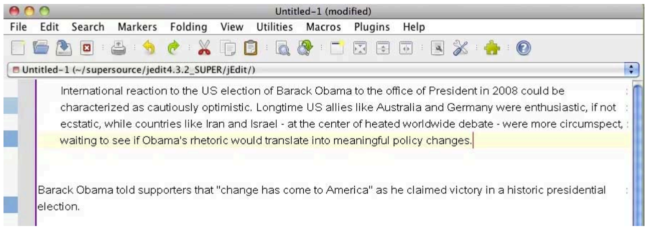
To understand how users visit web pages while coding, we organized a competition in which participants wrote a music visualization program in the Processing IDE in two hours. Four undergraduate Computer Science majors participated. The competition approximates end-user programmers' patterns of opportunistically learning to use new libraries [4]. It has two limitations: First, time pressure may have led participants to adopt different behaviors. Second, results will not generalize to professionals who work with the same set of technologies on a daily basis.

On average, participants visited 37 web pages (min: 26, max: 49); of which 23 were unique (min: 15, max: 30). Predictably, page visits included search engines and standard reference pages for the provided libraries. However, logs also contained pages on musical scales, frequency tables, discussions of color spaces and third party tutorials. Capturing these less obvious sources is a benefit of HyperSource over automatic documentation recommenders [6]. Associations were clustered on relatively few lines: 7 out of 141 lines of code were annotated on average (min: 5/102, max: 9/177). The relatively low number of highlighted lines suggests that browsing occurs in a small number of thematic chunks; and that association highlights can serve as useful indicators of "interesting" code sections.

Comparing Source Understanding

To assess how useful annotated source documents can be, we compared how participants answered questions about code with and without HyperSource. We manually constructed two HyperSource-annotated source files based on the music visualization code. These files likely overestimate the utility of our tool as we did not add irrelevant links. However, they can provide baseline data whether the approach benefits users. A set of four new participants reviewed each file and answered questions about the code for 15 minutes per file. Questions focused on the meaning of function calls, parameters, and the rationale for design choices in the code. Participants completed one task with HyperSource and one without. Afterwards participants reflected in writing on the two conditions.

Three participants explicitly commented that HyperSource helped them understand the unfamiliar code faster: *"It was much easier to answer the questions with the annotated URLs"*; *"a valuable tool to accelerate the process of reading code."* Two commented that the links provided the right starting point for understanding which set of sites were consulted, but that they still explored other (non-captured)


Untitled-1 (modified)
File Edit Search Markers Folding View Utilities Macros Plugins Help
Untitled-1 (~/supersource/jedit4.3.2_SUPER/jEdit/)

International reaction to the US election of Barack Obama to the office of President in 2008 could be characterized as cautiously optimistic. Longtime US allies like Australia and Germany were enthusiastic, if not ecstatic, while countries like Iran and Israel - at the center of heated worldwide debate - were more circumspect, waiting to see if Obama's rhetoric would translate into meaningful policy changes.

Barack Obama told supporters that "change has come to America" as he claimed victory in a historic presidential election.

Figure 6: Part of an essay written in jEdit + HyperSource.

pages based on that starting set. One participant pointed out that programming is a very keyboard-centric activity and that tools like HyperSource must therefore be accessible without relying on (slow) mouse interactions.

Writing an Essay With Automatic Reference Tracking

Writers of prose may also benefit from HyperSource. To inform our intuition about domains beyond programming, we asked an undergraduate English student to write an essay about international media reception of the Obama presidency in jEdit. This task required visiting many diverse web resources. Figure 6 shows an intermediate stage of this task, with multiple annotated lines of text.

To our surprise, the final document contained few such annotations. We hypothesize that writing prose is fundamentally about *re-writing*: our participant frequently edited and rephrased entire sentences. HyperSource associates web pages with individual characters; if those characters are removed from the document, e.g., through replacement of a sentence, the associations vanish as well. This suggests that more sophisticated tracking algorithms may be needed.

TRACKING MORE THAN WEB PAGES

HyperSource associates relevant information from sources external to a code document with regions within that code document. This principle can be generalized beyond web pages. Developers may also have to change their underlying system configuration for their project, *e.g.*, by editing system variables, or changing file permissions. Such tasks are frequently accomplished through shell commands.

We investigated whether capturing and associating shell command histories could be useful by implementing a custom version of the Bourne-Again Shell (`bash`). Our implementation hooks into the history mechanism: whenever an executed command line is added to the bash history, the command is also sent to a HyperSource editor. This experiment yielded two insights. First, a project-specific command line history can provide valuable context about actions previous developers took outside the IDE. Second, terminal commands have *weak locality* — they are not strongly related to specific code sections. It may therefore be more appropriate to associate command line histories with the directories in which the commands were executed.

RELATED WORK

CodeTrail [4] most closely matches our motivation: It automatically selects and attaches documentation to projects using known documentation sites; opens attached documentation in a browser; and enables users to manually bookmark URLs at the file level. HyperSource differs in focus: it tracks all visited Web pages implicitly and attaches web histories to specific sections of code. HyperSource also introduces a novel user interface for showing and interacting with annotations. Blueprint [2] incorporates interactions for finding and integrating example code directly into the IDE. In contrast, with HyperSource developers continue to use their Web browser.

Better code search engines, *e.g.*, Mica [9], integrate search for documentation and example source. Such tools are not aware of the developer's IDE. HyperSource is also related to research that records Web browsing activity, *e.g.*, ActionShot [7]. We might leverage insights on session summarization from this research. Finally, HyperSource is related to task-oriented development tools such as Mylar [6], that aggregate information from disparate sources.

FUTURE DIRECTIONS: RECOMMENDATIONS

In future work, we plan to investigate the additional benefit that can be realized by mining a collection of HyperSource-annotated source documents for *code-directed collaborative filtering*. We believe that analyzing the code structure of annotated lines and the URLs attached to those lines can yield useful recommendations which URLs are frequently visited for certain types and methods. Validating this intuition will require a larger deployment of HyperSource.

REFERENCES

1. Processing 1.0. http://processing.org/.
2. Brandt, J., Dontcheva, M., Weskamp, M., and Klemmer, S.R. Example-centric programming: integrating web search into the development environment. *Proceedings of CHI 2010*, ACM (2010), 513-522.
3. 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 CHI 2009*, ACM (2009), 1589-1598.
4. Goldman, M. and Miller, R.C. Codetrail: Connecting source code and web resources. *Journal of Visual Languages & Computing 20*, 4 (2009), 223-235.
5. Hill, W.C., Hollan, J.D., Wroblewski, D., and McCandless, T. Edit wear and read wear. *Proceedings of CHI 1992*, ACM (1992), 3-9.
6. Kersten, M. and Murphy, G.C. Mylar: a degree-of-interest model for IDEs. *Proceedings of AOSD 2005*, ACM (2005), 159–168.
7. Li, I., Nichols, J., Lau, T., Drews, C., and Cypher, A. Here's what I did: sharing and reusing web activity with ActionShot. *Proceedings of CHI 2010*, ACM (2010), 723-732.
8. Pirolli, P. *Information foraging theory: adaptive interaction with information*. Oxford University Press, 2007.
9. Stylos, J. and Myers, B.A. Mica: A Web-Search Tool for Finding API Components and Examples. *Proceedings of VL/HCC 2006*, IEEE Computer Society (2006), 195–202.
10. Teevan, J., Adar, E., Jones, R., and Potts, M.A.S. Information re-retrieval: repeat queries in Yahoo's logs. *Proceedings of SIGIR 2007*, ACM (2007), 151-158.