Codetrail: Connecting source code and web resources

Citation: Goldman, Max, and Robert C. Miller. “Codetrail: Connecting source code and web resources.” Journal of Visual Languages & Computing 20.4 (2009): 223-235.

Published Version: http://dx.doi.org/10.1016/j.jvlc.2009.04.003

Publisher: Elsevier

Permanent Link: http://hdl.handle.net/1721.1/51691

Version: Author's final manuscript: final author’s manuscript post peer review, without publisher’s formatting or copy editing

Codetrail: Connecting Source Code and Web Resources

Max Goldman and Robert C. Miller

MIT CSAIL

32 Vassar St.

Cambridge, MA 02139

{maxg, rcm}@mit.edu

March 16, 2009

Abstract

When faced with the need for documentation, examples, bug fixes, error descriptions, code snippets, workarounds, templates, patterns, or advice, software developers frequently turn to their web browser. Web resources both organized and authoritative as well as informal and community-driven are heavily used by developers. The time and attention devoted to finding (or re-finding) and navigating these sites is significant. We present Codetrail, a system that demonstrates how the developer's use of web resources can be improved by connecting the Eclipse integrated development environment (IDE) and the Firefox web browser. Codetrail uses a communication channel and shared data model between these applications to implement a variety of integrative tools. By combining information previously available only to the IDE or the web browser alone (such as editing history, code contents, and recent browsing), Codetrail can automate previously manual tasks and enable new interactions that exploit the marriage of data and functionality from Firefox and Eclipse. Just as the IDE will change the contents of peripheral views to focus on the particular code or task with which the developer is engaged, so, too, the web browser can be focused on the developer's current context and task.

1 Introduction

To suggest that the web is a source of information for software developers is perhaps trite. The web connects developers to language and API documentation, code examples, bug reports and workarounds, tutorials, articles, code ranging from one-liners to sketches to pages of boilerplate, and innumerable sources of discussion and debate on programming systems popular and obscure. Since the domain of web resources ranges from weblog posts and forum discussions to web-accessible interfaces for a development team's own bug database or version control, there is no one way to characterize web resources or shoehorn them into a developer's existing workflow.

Once new tools such as the web prove their worth in the developer's toolset, they become integrated into the development environment. The Eclipse integrated development environment provides ready evidence, with specialized plug-ins for integrating with version control systems, bug databases, and many other tools. But integration with web resources remains lacking.

Software developers already use a powerful tool for accessing the web: their browser. Web browsers are customized and configured according to the user's information needs. The browser stores bookmarks, cookies, certificates, and browsing history. Custom style sheets and font choices may be applied. In the case of Firefox, an extension mechanism enables users to install components that provide additional customizations, or that allow programming-savvy users to construct new customizations of their own. Compare this rich interface against the built-in web browsing component that ships with Eclipse. The latter provides none of the above features, and it is consequently the authors' experience that this browser is rarely used. Instead of a simple HTML renderer with minimal UI, software developers need a full-featured web browser to be integrated into their working environment.

Our prototype system, Codetrail, integrates the web browser and the IDE via a shared communication channel and data pool, and provides new tools that help the developer integrate web resources into the programming workflow and into the project itself. Of particular interest is the way Codetrail can enable the web browser to act as an IDE *view*, where the contents of the browser are automatically focused on the code the developer is currently editing. Many other Eclipse tools behave in this fashion.

Codetrail aims to eliminate the manual effort currently spent coordinating IDE and web browser in order to make use of web resources. By making previously manual tasks automatic or previously laborious tasks straightforward, the developer remains focused on software development. Both developers and project teams gain the additional benefit of more comprehensive and organized records of the online resources used during the development process.

In the balance of this paper we report on the results of an informal field study on developers' web use, describe in detail a number of tools implemented on the Codetrail platform, discuss the results of a user study in which developers worked with Codetrail, and then describe general requirements for a useful integration mechanism as well as specific elements of our implementation. Codetrail was first presented in [2], appearing here with elaboration on technical details and with additional evaluation.

2 Related Work

Programmers' working environments have long been constructed by combining a variety of tools [3], and this practice has been examined by many researchers (e.g. the case study of [19]). In the classification of IDE tool integration from [20], Codetrail falls in the most powerful class, synchronizing the activities of Eclipse and Firefox using a loosely-coupled architecture.

Developers' information needs have also received study. Much of the historical information about code critical to developers' work is implicit [11], and Codetrail is an attempt to make some of that knowledge explicit and discoverable. Given the results from [9], which suggest that programmers often turn to their colleagues for help, we contend that giving those programmers tools to connect with web resources may be profitable. In an earlier examination of design requirements for maintenance-oriented IDEs [8], developers in a user study spent 11% of the total task time either reading the Java API or switching between applications such as their IDE and web browser, suggesting that even for relatively small tasks, documentation is important and there is overhead to be eliminated.

A variety of systems have been proposed that attempt to help developers use the web more effectively. Assieme [5] augments search results with code excerpts, and improves search result accuracy by combining a web crawler with a compiler that can identify implicit relations between code. Mica [16] is an earlier system which also augments search result display. Such

tools certainly do increase the usefulness or accessibility of web resources, but they do not integrate those resources into the development environment.

Other projects have integrated new tools into the IDE. Hipikat [17, 18] tracks project memory and recommends resources to newcomers, including web resources. Jazz [1] integrates presence and instant messaging to connect coworkers. Mylyn (previously Mylar) [6] is based on the notion of a task-focused user interface in the IDE. It provides integrated access to some of the same resources that might be linked using Codetrail, but does so using its own interface. Our focus is on what the web browser, as a flexible and familiar tool for accessing information on the web, brings to the IDE.

3 Code and the Web

In order to better understand the nature of developers' interaction with web resources, we conducted a small informal field study that examined the development-related browsing behavior of research group members. Four subjects participated for between one and three weeks each by installing a tool that recorded information about files edited in Eclipse and web pages loaded in Firefox. All were actively involved in software development at the time. Before submitting their data, participants had the opportunity to "scrub" the log of any web site visits they considered sensitive and did not wish to report, and no record of the number or nature of these scrubbed sites was retained. In total, 6,409 page loads were reported.

The URLs and contents (where accessible) of the pages were examined and each page was hand-labeled according to whether it was self-evidently *development related*, e.g. because it contained code or described programming tools. As a result, the count of development related pages may be an underestimate, if participants visited sites whose connection to software development was non-obvious or only by virtue of the problem domain. Overall, 10% of reported page loads (646) were obviously development related.

Since we are interested in usage of the web browser and IDE together, we considered events in temporal proximity. For page loads that occurred within a 5-minute window of saving some code file, 23% were development related.

Development related pages took a variety of forms, described below and enumerated in Table 1:

Searches Searches of all kinds were performed almost exclusively using the Google web search engine[1].

Code elements Example code element searches include "java.lang.math," "JSplitPane," and "setBorder," where the searcher is using the name of some language element.

Code ideas Example code idea searches include "php xsl" and "read from input java," where the searcher is looking for implementation-level information but has not identified particular code elements.

Sites Rather than discover information sources using queries for specific entities (e.g., by code element search), search queries were sometimes used to locate the sites themselves. Examples include "java 1.5 api" and "xulplanet." It is likely that these searches are a re-finding mechanism for developers who already know about the sites but choose not to use bookmarks or memorize URLs.

Tools An example tool search is "subversion branch," where the subject of the search is a component of the developer toolchain, not the program code.

Concepts At a higher level than code elements or tools, searches were observed where the subject was conceptual, related to general computational ideas. Examples include "fibonacci heap" and "halloween problem."

Documentation Documentation sites exhibit careful structure and are organized by documented element.

Official Often results of code element searches led users to official documentation, such as the Java API documentation provided by Sun[2]. Other examples include Facebook's API documentation[3] and Adobe's documentation on Flex[4].

Unofficial Not all documentation is maintained by the originator of a framework or system. Examples visited include XUL Planet's documentation on Mozilla APIs[5] and W3Schools' web technologies references[6].

Other resources Non-documentation resources.

[1] http://www.google.com/
[2] http://java.sun.com/reference/api/
[3] http://wiki.developers.facebook.com/index.php/API
[4] http://livedocs.adobe.com/flex/3/langref/
[5] http://www.xulplanet.com/references/
[6] http://www.w3schools.com/

Project Some visits were to project-related resources maintained by developers on the project; a bug database and web frontend for a version control repository are examples.

Navigation Example navigation, or portal, pages include the Eclipse and Sun home pages.

Content Most web information sources are not organized as documentation. Technical articles and tutorials, blogs and other personal web pages, wikis, forums, and mailing list archives were all visited by participants.

Note that the particular population of developers who reported results heavily influences the numbers in Table 1. All did at least some development in Java, increasing the relative representation of official documentation, since Sun provides extensive API documentation on the web. Participants were also working on smaller, loosely-organized academic projects, decreasing the relative number of project resources used in comparison to a large industry project with infrastructure to manage features and coordination.

This taxonomy motivates a general two-pronged approach to connecting code and the web in Codetrail. First, site searches and code element searches are targeted for automation, by creating stronger links between identified sources of official and unofficial documentation and the code. With these links, the user no longer needs to turn to a search engine. Second, connections between code and other useful web resources are facilitated by a general-purpose bookmarking mechanism that is less automated, but significantly more straightforward than currently available.

4 Applications

Having revealed some aspects of developers' web behavior, we now describe several tools we have implemented in Codetrail, considering usage scenarios and relevant design principles. The variety of tools serves to demonstrate that the system is a robust and flexible integration mechanism. We begin by describing how Codetrail automatically integrates structured documentation sites, first in the context of automatically identifying and browsing to Javadoc documentation (Sections 4.1 and 4.2), and then more generally in 4.3. Section 4.4 turns to integration with unstructured resources via user-created bookmarks, and in Section 4.5 we discuss automatically-created bookmarks in the case of copied code.

All of these tools were designed using an iterative process. Early versions of the system were deployed to students and faculty in our research group, who provided feedback on the usefulness and usability of Codetrail's features.

4.1 Automatic Javadoc Attachment

Eclipse already includes a mechanism for "attaching" Javadoc URLs to libraries, although the user interface for this feature is somewhat obscure. Attached Javadocs are used to display relevant documentation for the code element (class name, method name, etc.) under the editor cursor in a separate view, and to add information to code completion suggestions (Figure 1). While often useful, the amount of information presented and visible at any time is limited. The separate view, in particular, is an impoverished window on the Javadocs, with no ability to navigate to related methods or classes, or to see any context for the current artifact.

Codetrail bypasses the obscure Javadoc attachment UI by automatically identifying relevant Javadoc documentation when the user browses to it in Firefox. Documentation sites generated by Sun's `javadoc` tool have a fixed form and include a machine-readable file enumerating the packages that the site documents. Codetrail downloads this file and searches the active Eclipse projects for libraries containing those packages. When a match is found, the user is prompted as in Figure 2. Since the matching process takes trivial time, the prompt appears almost immediately after the user loads the Javadoc site.

If the developer instructs Codetrail to attach the Javadocs, the existing Eclipse documentation attachment mechanism is used. The benefits are immediate in improved tool tip and auto-complete information, and the existing Javadoc view will now become active. Further benefits will be described in the sections to follow.

The user is prompted to confirm documentation attachment so that the system's operation is visible: our goal is to have Codetrail improve workflow and add new functionality without making the development environment more inscrutable. By presenting a prompt, we provide the developer with a mental model of how the system operates: "when I visit a documentation page, Eclipse asks me about attaching it." Without this model, a user might not make the connection between visiting documentation in the browser and subsequent links to that documentation in Eclipse. Prompting does introduce a risk of annoyance, which we mitigate in

several ways. Only one prompt is offered per Javadoc site, and only when matching packages are found without an existing documentation attachment, so we expect users to find the dialogs infrequent and timely. The prompt is also non-modal and the user may choose to disregard it, in which case the dialog is dismissed when the next web page is loaded.

4.2 Automatic Browsing to Documentation

While automatic linking of Javadocs to library code improves the developer's experience, we have not yet resolved the aforementioned issues with the existing ways to access that documentation in Eclipse. Rather than rely on an in-IDE view with limited screen space and functionality, Codetrail uses Firefox to automatically browse to the full documentation for code elements under the editor cursor. Automatically browsing to documentation addresses the large number of queries for Java package, class, and method names seen in the initial field study. It also reduces queries for information sources themselves, as developers need only click on a code element associated with a given documentation source in order to reach that source and begin navigating within it.

Codetrail implements automatic browsing by creating, as needed, a special browser tab in Firefox (Figure 3). All automatic browsing occurs in this tab, which consistently displays an identifying label and Eclipse icon. The user is free to browse within this tab, achieving the important capacity for exploration of other documentation related to a particular code element. While the current implementation assumes that only one documentation item is relevant for a given element (in line with Eclipse's model), the user could be presented with a choice of multiple documentation sources – or, multiple browser tabs could be used to put all sources at the ready.

Automatic browsing is our first example of treating the web browser as an IDE *view*, a particular integration mode where the browser becomes subordinate to the IDE editor. Operating as a view, the browser displays information relevant to the code artifacts on which the developer is currently focused. This allows the developer to immediately access full documentation for every class, method, or other keyword the cursor touches, just as views included with Eclipse give instant information about other properties like location in a class hierarchy.

4.3 Generic Documentation Attachment

Javadoc documentation is only one form of the many kinds of official and unofficial documentation developers may use. We would like to allow developers to associate any useful source of documentation with their project and receive the same benefits of immediate access.

Many documentation sources exhibit significant structure, even if that structure is unique to the particular source. XUL Planet is an excellent example, with an individual page documenting each interface in the Mozilla API.

Consider the scenario in which a developer is working on a project and visiting documentation sites as he works, looking up code elements related to the project. We would like to automatically learn the association between keywords appearing in the developer's code and pages on the documentation site, without having to ask the user to describe that connection. Codetrail achieves this automated documentation identification by observing both code contents and browsing history.

By storing a record of recent browsing history, Codetrail can identify common URL patterns that differ in only a particular component. For example, suppose a user discovers the XUL Planet site via a Google query, visiting URLs:

www.google.com/search?q=nsifile

www.xulplanet.com/references/xpcomref/ifaces/nsIFile.html

www.xulplanet.com/references/xpcomref/

www.xulplanet.com/references/xpcomref/group_RDF.html

www.xulplanet.com/references/xpcomref/ifaces/nsIRDFNode.html

www.xulplanet.com/references/xpcomref/ifaces/nsIRDFLiteral.html

Codetrail will extract the pattern:

www.xulplanet.com/references/xpcomref/ifaces/<*keyword*>.html

The values of differing components are used as keywords in a search through the user's Eclipse projects – in this case, nsIFile, nsIRDFNode, and nsIRDFLiteral. Matching keywords in project files (in this case at least two of the three candidates would have to be found) is strong evidence that the site contains relevant documentation. If a match is discovered, the user is shown a prompt similar to the one used for Javadocs, although here we identify the documentation link by keywords instead of library name, since we do not know anything about

the semantics of these keywords (Figure 4). While system visibility is important as before, here the prompt is additionally necessary because we are using documentation identification heuristics that may generate some false positives. With the user's approval, Codetrail uses the Yahoo! web search API[7] to retrieve pages matching the same URL pattern, additionally restricting matches to pages whose titles contain words that were common to the original pages used to identify keywords. Continuing our example, the query here would be:

site:www.xulplanet.com inurl:references inurl:xpcomref inurl:ifaces

And the results returned would include:

www.xulplanet.com/references/xpcomref/ifaces/nsIUpdate.html

www.xulplanet.com/references/xpcomref/ifaces/nsIURI.html

... 900 more ...

From this set of URLs, the set of keywords to associate with this documentation source is extracted and stored. The same automatic-browsing technique is applied when the user moves their editing cursor over a known keyword. So now, if our example user clicks on "nsIURI" in the Eclipse editor, XUL Planet documentation for that interface will come up in Firefox, without the programmer ever having visited that particular page before.

As with many web structure extraction heuristics, an algorithm of this sort requires significant development effort to work across many sites, each with its own particular quirks. The current Codetrail implementation has been tested successfully on several sites, including XUL Planet, the MySQL Reference Manual[8], and documentation sections of the Mozilla Developer Center[9].

4.4 Bookmarks

Not all associations between code and the web link to organized resources that can be detected automatically. In order to handle arbitrary links, the system provides a flexible bookmarking mechanism that the user can invoke to link any web location to any code file. Some developers may already create a weak version of such links by including URLs in comments in source code files. However, creating links for code files alone is, in our estimation, too restrictive. The

[7] http://developer.yahoo.com/search/web/

[8] http://dev.mysql.com/doc/

[9] http://developer.mozilla.org/en/docs/

tool searches observed in the initial field study might yield resources valuable not in reference to a specific file or files, but to the project as a whole, or to some part of it. Build scripts, configuration files, to-do lists, and binary libraries, for example, all might have web resources relevant to them. Codetrail stores bookmarks as metadata, outside the target file, so that developers can create bookmarks in files of different languages without requiring different syntaxes; in files whose language may not have a comment syntax, or have any particular syntax at all; on folders; and on binary files and other unmodifiable resources.

Our approach is to create *bookmark files* in the Eclipse project to hold the bookmark associations. Since there is no popular cross-platform web link file type, we use a simple XML schema that also conforms to the specification for Apple web location (webloc) files. In both Firefox and Eclipse, Codetrail presents interfaces for accessing bookmarks, and for creating them by associating web locations with files or directories (Figure 5). Equivalent functionality in both applications keeps bookmarking close at hand and eliminates application switching to record an association. Codetrail enables these interfaces by giving Firefox access to the Eclipse file hierarchy, and Eclipse access to recent Firefox browsing. Bookmarks for the currently-edited file are displayed in an Eclipse view shown in Figure 5c. This view shows bookmarks directly associated with the current file at the top, but keeps in view all bookmarks attached to containers of that file up the resource hierarchy (in the figure, for example, a link to the Eclipse FAQ is visible from any file in the "plugin" directory of the project).

Storing bookmarks as metadata files additionally allows the developer to organize them independently of the target code or other files or directories and provides a straightforward mechanism for sharing them with other developers. Codetrail allows bookmark files to be stored anywhere beneath a designated bookmarks directory in the project, meaning that a developer or development team can organize them according to their needs: tutorials in one folder, references in another; resources for newcomers to the project who need to orient themselves combined in one place; arranged by source web site to track the sources of input to a project; and so on. Whether organized or not, storing bookmarks as files also allows them to be checked in to the project's version control system, providing a clear path for sharing bookmarks with other team members.

4.5 Code Snippets

Although the design of the bookmark tool emphasizes flexibility, creating bookmarks to code-bearing web pages – tutorials, templates, examples, snippets of code posted in forums – may still be a common case. In this scenario, Codetrail can automate the process by identifying when code is copied from the web into a source file. Our approach is not limited to direct copy-and-paste actions, since our observation suggests programmers will often copy by hand in order to better understand the code, or to avoid introducing errors when undesired or incorrect pieces of their source are incorporated into the target. Instead, Codetrail examines all newly-written code as it is saved to disk, and compares those pieces of source code to potential code blocks on recently-browsed web sites. Those potential blocks are currently identified using a simple heuristic that searches for HTML `pre` elements, corresponding to pre-formatted text, often used for displaying source code with a conventional fixed-width font. Correspondence between saved and browsed code is determined using the Smith-Waterman algorithm for local sequence alignment [15]. Code matches are classified as token sequences representing at least 25% of the changed segment and of the potential web source with an alignment score of at least 5 (gap penalty of 1). When a such a code match is identified, a bookmark is automatically created.

Automatic creation of bookmarks to record the source of copied code serves to retain bread-crumbs that can later allow developers to answer questions like "why did I choose to write the code this way," or "that variable name looks wrong, is it a bug?" Knowing the provenance of the code can also help explain what it does and how it works, since the web page from which code was adapted may offer more context, discussion, and caveats.

5 Discussion

With the tools in the previous section, we have demonstrated a few different automated means of identifying that project and web resources are related: by finding tokens common to code and URLs (for generic documentation attachment) or to code and web page contents (copied snippet identification), and by language-specific mechanisms (Javadoc attachment). Other techniques could be employed as well; for example work on read wear [4] could be applied to identify related resources by their frequent use during development sessions. In deciding

how to automate identification of relevant resources, a primary design constraint was that the technique must reveal not only the fact that a resource is related, but also the nature of their relationship. For generic documentation attachment in particular, this means that our approach enables Codetrail to link keywords in the user's code to matching documentation web pages even if those keywords never before appeared in the code, and the particular web pages have never been visited, as long as the use of other keywords previously led to the discovery of the relevant web site.

For recording the relationship between code and web resources, we have already suggested that current practice often takes the form of URLs in source code comments, and outlined how this is often insufficient. Links as URLs in the source code also fail in the case of documentation attachment: since a site may document many keywords appearing throughout many source code files, there is no single best location in the source to indicate the relationship. Thus, we prefer a strategy of recording relationships with an independent representation that can be stored and shared apart from source code.

6 User Study

In order to evaluate the tools implemented in Codetrail, we conducted a user study with eleven participants, four of whom are in our research group. Users downloaded and installed a version of Codetrail that was instrumented to record a variety of programmatic and user interface events in both Eclipse and Firefox. The system was installed by participants on their own computers. Before using Codetrail, users provided optional demographic information and were given the opportunity to read descriptions or view short screencasts demonstrating some of Codetrail's features: documentation attachment, automatic browsing to documentation, and creating bookmarks.

Usage logs collected from the study were periodically reported automatically as long as participants continued to use the plug-ins. Participants were also sent a post-study survey via email, to which seven replied; this survey solicited their feedback on Codetrail and probed for possible improvements to the system.

All of the user study participants were male, most in their 20's or 30's. All indicated that they do software development in Java; Python was the second most often mentioned language.

Three identified themselves as professional software developers; the rest identified as students or researchers. We did not request specific details of participants' current projects or tasks.

Participants used Codetrail for an average of 17.6 days (σ=11.9), not counting days during which no events were recorded in either Eclipse or Firefox (e.g. for installations on a professional developer's office computer). We counted as *active development time* all time intervals within one minute of a user interface event in Eclipse (selecting windows, saving files, etc.). Participants averaged 24.6 hours of active development during the study (σ=27.9).

During that development time, each user browsed to an average of 74.5 URLs per hour (σ=58.7). This number may have been influenced by the presence of Codetrail, but supports our assertion that the web is an important resource for developers.

6.1 Automatic Browsing to Documentation

We tracked usage of the automatic browsing to documentation feature (Section 4.2) by logging when users activated the browser tab created by Codetrail to show documentation, and when they gave the Firefox application focus with that tab foremost. Nine of the eleven participants used this feature; those who used it did so an average of 7.9 times per development hour, viewing an average of 3.1 different documentation pages per hour.

In survey feedback, participants who used this feature were unanimously positive:

"I used this feature often. I found it very useful" (User 4).

"I used (and still do) this feature all the time. This is one of the most useful features that I've encountered in Eclipse ever since I started using the IDE" (6).

"It's much better than popup dialogs/hovers which obscure the code" (11).

Some users suggested further automation of the browsing, for example: *"I did use this feature, and I found it useful. I feel it would have been nice if some hotkey would put the focus on Firefox, and bring the special tab to the front"* (7). And one user found it problematic that Codetrail would always update the tab to the documentation of the identifier under the Eclipse editing cursor, even if he was still using the previous page: *"... it was too quick to browse *away* from documentation that I still needed"* (10).

Additionally, participants' use of the automatic browsing feature was almost exclusively with Javadoc documentation sources that were already known to Eclipse. These sites included:

- Java 5.0 API (http://java.sun.com/j2se/1.5.0/docs/api)
- Java 6 API (http://java.sun.com/javase/6/docs/api)
- JUnit 4.3 API (http://www.junit.org/junit/javadoc/4.3)

Use of the features to automatically identify and attach sources of Javadoc and generic documentation (Sections 4.1 and 4.3) were limited: four users elected to attach Javadocs at least once, and none used the generic attachment mechanism.

For Javadoc attachment, a total of nine attachment prompts were shown to those four users, who instructed Codetrail to do the attachment every time. URLs attached included:

- New JSR 166 Java 7 APIs
  (http://gee.cs.oswego.edu/dl/jsr166/dist/jsr166ydocs)
- JavaMail email & messaging API
  (http://java.sun.com/products/javamail/javadocs)
- JUnit API, current version (http://junit.sourceforge.net/javadoc)
- ImageJ image processing & analysis API
  (http://rsbweb.nih.gov/ij/developer/api)
- A proprietary API
  (file:/C:/build/dev[...]/[...]Framework/main/javadoc)

All but the last of these URLs represent the documentation for Java libraries used by participants, precisely the use case for which this feature was intended. The last URL represents an instance in which the user has attached documentation for libraries developed internally to his employer. Regarding these attachments, one participant called the feature *"very useful"* (8); another, his *"preferred way to attach online documentation to an existing project"* (4).

In contrast, while 86 generic attachment prompts were shown to eight users, none were affirmed. This large number is due in part to a bug in the version of Codetrail initially used in the study, which failed to remember negative answers and would prompt users again on subsequent visits to the same site.

User feedback was lackluster, and several users indicated that documentation sources were not accurately identified:

"I noticed this feature, but I found it sometimes misinterpreted my behavior" (2).

"I think I used this [feature] once to attach a Javadoc, and I found that useful, but the system was never correct when it asked me to link any other sort of documentation..." (7).

At least some users were also unsure about how the feature would work, being familiar with Javadocs but unsure how other documentation sources would be integrated with Eclipse: *"At the end, I don't believe I ended up attaching any of the pages. I think the most likely reason was that, even if I had attached any page, I wouldn't know where to look for it"* (6).

We conclude that the automatic browsing feature of Codetrail, and the use of Firefox as an IDE view for documentation, will be appealing and useful to a large population of developers. Automatically attaching documentation is convenient, but more work is needed to make it effective with documentation that does not provide a machine-readable description of the code elements it documents. The heuristics must be more accurate, taking into account more than URL keywords; and the consequences of attachment must be clear to developers, perhaps with a more descriptive attachment prompt and immediate feedback in Eclipse.

6.2 Bookmarks and Code Snippets

The bookmarks feature (Section 4.4) was largely unused. Five of the eleven user study participants created one or two bookmarks; the rest did not use the feature. Nearly all of those bookmarks appear to be created in order to test or explore the Codetrail interface, and none meet the subjective self-evidently development-related test used in the initial exploratory study (Section 3).

Although users were asked to describe why the feature was not useful or how to improve it, they offered limited feedback. One user made reference to the common pattern of re-finding as an alternative to bookmarking: *"I don't use bookmarks much... I generally just search again"* (User 7). Another participant wrote that he *"didn't use this feature actively. I put some URLs in source code comments... however, because that's how I habitually include links..."* (10).

These comments suggest that web bookmarks as implemented were not in line with our

participants' development practices. Other survey feedback pointed to a lack of knowledge of this feature in Codetrail: *"I wasn't really sure how to create bookmarks or where to look for them inside Eclipse"* (6). This feature of the system was the only one in which participants had to explicitly enable an Eclipse view in order to make use of it. Since seven of the eleven participants did so, we conclude that while the bookmarks interface could certainly be improved, discoverability was not the primary barrier to uptake, and that our model of how bookmarks link code artifacts to web resources may require revision.

Codetrail also automatically identified 33 snippets of code duplicated by participants and created bookmarks for them (Section 4.5). Six participants logged at least one snippet. Based on informal discussions with three of the study participants accessible to us, some of these snippets indeed represent code copied from Java API documentation pages, including:

- Exception handling
  (http://java.sun.com/javase/6/docs/api/java/lang/Throwable.html)
- Cache flushing
  (http://java.sun.com/javase/6/docs/api/java/util/LinkedHashMap.html)

In other cases, the snippets represent instances where the user was viewing in Firefox part of the same source code he was editing in Eclipse, either because the code was being posted online, or because some code was reproduced in the output of an automated build system.

Without anecdotes describing the use of this feature in its primary intended mode – copying source code from example or tutorial pages – and since URLs and matching code segments of participants' suspected code clones were not collected for privacy reasons, it is difficult to evaluate this feature. Most users said they had not noted or used the feature. One was positive: *"I saw those bookmarks created in one case, and I was happy because they were the right ones"* (10). Another opined that *"... the code copying stuff would be useful if an organization had a library of such snippets accessible via a web front-end – which is not the case here"* (11), a reminder of how this feature may need specialization for use in a closed-source setting. Overall, we conclude that these bookmark-related features of Codetrail need more refinement to become useful. In particular, bookmark storage in source code, as opposed to metadata files, merits examination.

To close this user study discussion, we note that ten of the eleven participants continued to have Codetrail installed and enabled for at least a week after the study ended, based on continued submission of usage logs. Of those, seven participants made use of the automatic documentation browsing feature, including one who had not used that feature during the study period itself. We take this as strong evidence that, for at least some users, the system provides substantively useful integration of the IDE and web browser.

7 Infrastructure

Both Eclipse and Firefox provide plug-in mechanisms for extending their functionality, and Codetrail uses these facilities. The design of Codetrail was motivated by the following goals:

Shared data Both Firefox and Eclipse are powerful computation engines in their respective domains, but most essential to Codetrail's operation is the sharing of data between them. Much of Codetrail's power is derived from combining data produced by or (previously) known to only the individual applications. At least session-long persistent data is a requirement to enable, for example, heuristics that rely on the user's recent web browsing or code editing history.

Publish-subscribe Components in both applications must be able to listen for patterns of data produced by other components and take appropriate action in response.

User feedback In order to facilitate a responsive user interface, components must be able to present user feedback at the source of an event. Immediate at-the-source feedback is important especially when the user should be prompted about their current web browsing location, and we wish to present that prompt while the user is still focused on that site.

Modularity As a research system, the ability to add new modules to the system, rapidly explore new ways of combining data and functionality, and keep dependencies between modules at a minimum was necessary. Modular design is also important to users, so that individual pieces of functionality can be enabled or disabled by developers who find them more or less useful.

Cross-platform We require a cross-platform system that works for Eclipse and Firefox users

on all three of the major operating systems – Windows, Mac OS X, and Linux – where they are used.

The implementation of Codetrail uses Java in Eclipse and JavaScript in Firefox. A simple socket protocol is used for communication, and the pool of shared data is represented by an RDF graph (Resource Description Framework, in which data are modeled as *(subject, predicate, object)* triples forming a directed labeled multigraph) [10]. The graph is managed in the Eclipse JVM using the Jena RDF framework [12]. It is exposed in Firefox according to the existing Mozilla interface for RDF datasources. The use of RDF in a tuple space is similar to other work on semantic tuple spaces [7, 13].

The functionality of Codetrail is divided into separate modules. The system maintains a set of currently-activated modules, where each module implements some integrative functionality, and is coupled to other modules only by virtue of the data it produces or consumes and the schema of those data. The user has control over which modules are active in order to selectively enable functionality useful to them.

7.1 Shared Data

Codetrail modules communicate and coordinate by adding RDF triples to the shared graph, and subscribing themselves as listeners for patterns of new triples. For example, one module we have implemented tracks the user's web browsing and adds to the graph structures with data about each page load (such as the URL). Another module is concerned with automatically identifying Javadoc sites in order to attach them in Eclipse. This module listens for updates from the browsing module, identifies Javadoc sites, and takes the appropriate actions.

Figure 6 shows the evolution of the shared RDF graph over time with these two modules in operation, based on triples added by the browsing module. Initially (6a), we suppose there are structures in the graph corresponding to two URLs, perhaps from previous browsing activity. One of these URLs is in an *#ignore* relationship with *#mod-javadoc-attacher*, indicating that while it is a Javadoc site, the user has previously elected not to attach it. If the user browses again to that site, the graph is extended as in Figure 6b with a structure recording the page load. Browsing to the second, non-ignored site, would add the structure in 6c.

Adding to the graph is straightforward, but subscribing to notifications about those addi-

tions is more complex. To subscribe to updates, graph listeners specify both a *template triple* and an *anchored query*. The template triple takes the form *(subject, predicate, object)*, where each of *subject*, *predicate*, and *object* may be null. This template is compared to new triples added to the shared graph, with nulls acting as wildcards.

When the template matches a new triple, the listener's anchored query is executed. This query uses the SPARQL language for querying RDF graphs [14]. It is *anchored* because it may include a particular clause that assigns variables in the query to elements of the newly-added tuple which matched the template and triggered query execution. If the results of executing the query – which may include additional filters – are nonempty, they are forwarded to the listener's callback method. This anchored query mechanism enables listeners to extract relevant information from the updated graph without having to filter out matches from old data, combining the expressive power of a relational query with the efficiency of template matching for examining new triples. Template triples alone cannot match a significant amount of data, so listeners would inevitably have to run queries to select out more. Subscription with queries alone, on the other hand, would require the system to re-run all queries after every change to the graph, and to maintain a record of what data had already be returned by a given query so as not to report it again.

Returning to our example, the automatic Javadoc identification module of Codetrail is a listener with template triple *(null, #action, #load)*, which matches new triples representing events whose *action* is a web page *load*. This template triple is illustrated in Figure 7a.

The Javadoc module also specifies an anchored query (Figure 7b) that filters matching page load events to those for *allclasses-frame.html*, a file name characteristic of Javadocs, and which are not for URLs previously blacklisted by the user. The query further selects out just the pertinent properties of the page load event to return to the listener: the source of the page load (that is, which instance of Firefox) and the URL. The SPARQL query is written as:

```
1 SELECT ?source ?url WHERE {
2   GRAPH :match { ?s ?p ?o }
3   { ?s :source ?source .
4     ?s :artifact [ :url ?url ] .
```

```
5       OPTIONAL { ?j :ignore ?url } .
6       FILTER regex(?url, ".*/allclasses-frame.html") .
7       FILTER ( ! bound(?j))
8   } }
```

In the first page load of Figure 6a to 6b, the template triple will match *(#event-1, #action, #load)*, and the anchored query will be executed. Line 2 of the query will anchor the results to this matched triple, and the query will return no results, because *#mod-javadoc-attacher* is bound to variable *?j* in line 5. Thus, the blacklisted URL is ignored. In the subsequent visit (6b to 6c), the template triple will anchor the query on *(#event-2, #action, #load)* and a result containing the *source* and *url* will be forwarded to the Javadoc module.

Finally, in addition to modifying and observing the graph, some Codetrail modules record pieces of the shared graph as XML files in the user's Eclipse projects: bookmarks and generic documentation specifications are stored in this way. Changes to these files are synchronized with the in-memory representation so that the effects of new files (e.g., a bookmark just received via version control update) and power users' manual edits are immediately reflected in Codetrail's behavior.

8 Future Work

Codetrail is a working system that has been developed with the feedback of users inside and outside our research group, but future work remains. In the system itself, we plan to continue adding new modules and improving existing ones, with a special focus on scalability that will allow Codetrail to work efficiently with large Eclipse projects. A number of components, including generic documentation identification and copied code tracking, use heuristics that can also be improved with further testing.

Although we have presented some preliminary data on developers' web usage habits, more remains to be learned. Our user study of Codetrail has shown that while the automatic browsing feature in particular was appreciated, there is more to be understood about how to support programmers' use of web resources, particularly with respect to bookmarks. And while the user study demonstrated subjective satisfaction on the part of the participants, we hypothesize that Codetrail will improve software developers' measurable effectiveness as

well. Specifically, we predict that Codetrail can reduce the amount of time developers spend navigating to web pages and the amount of time they spend re-finding previously-visited pages, as well as the cognitive load of these activities; and that these reductions will lead to increased performance in code understanding, authoring, and debugging tasks. Empirical evaluation of these hypotheses with a controlled laboratory or field experiment remains to be undertaken.

The ultimate resources in the software development process are people, and another area for future work is Codetrail-supported collaboration and communication between developers, further exploiting existing channels such as project version control, or creating new ones.

9 Conclusion

In this paper we have presented a taxonomy of software developers' web usage as motivation for integrating the IDE and web browser, as well as an architecture for achieving that integration, tools that can be built using it, an implementation of the entire system, and an evaluation of that implementation.

Web resources of a wide variety are important to the programmer's development process, and Codetrail provides both specific mechanisms to automate previously manual interactions with web resources, as well as general mechanisms to integrate those resources into the development environment where it was previously tedious. The several tools described here, including automatic detection and connection of documentation and the creation of links, or bookmarks, from code to relevant web sites, demonstrate the advantages of bringing the user's full-featured web browser into the IDE as a participant in the development process. Treating the browser as a view, by automatically loading content relevant to the source code currently being edited, is one notable instantiation of this idea.

The documentation tools in Codetrail aim to satisfy the important goal of bringing needed information to the developer as quickly and fully as possible. The more general bookmarking function helps improve the collective memory of a software project by keeping alongside the code a record of resources that helped shape that code.

Developers don't need a watered-down web browser in their IDE; they need their IDE to integrate with the web browser they already use and have customized to their liking. Unlike

many of the tools in an IDE, the web browser is an application platform in and of itself, and must be treated as a first-class citizen in any integration scenario before developers will be satisfied. Codetrail demonstrates how this combination may be achieved with Eclipse and Firefox.

Acknowledgements

We thank members of the UID group for their valuable feedback and all study participants for their time, cooperation, and feedback. This work was supported in part by the National Science Foundation under award IIS-0447800, by Quanta Computer, and by DARPA. Opinions, findings, conclusions, or recommendations expressed are those of the authors and do not necessarily reflect the views of the sponsors.

References

[1] L.-T. Cheng, S. Hupfer, S. Ross, J. Patterson, B. Clark, and C. de Souza. Jazz: a collaborative application development environment. In *OOPSLA '03*, pages 102–103, 2003.

[2] M. Goldman and R. Miller. Codetrail: Connecting source code and web resources. *Proc. of VL/HCC '08*, pages 65–72, 2008.

[3] A. Habermann and D. Notkin. Gandalf: software development environments. *IEEE Transactions on Software Engineering*, 1986.

[4] W. C. Hill, J. D. Hollan, D. Wroblewski, and T. McCandless. Edit wear and read wear. In *CHI '92*, pages 3–9, 1992.

[5] R. Hoffmann, J. Fogarty, and D. S. Weld. Assieme: finding and leveraging implicit references in a web search interface for programmers. In *Proc. of UIST '07*, pages 13–22, 2007.

[6] M. Kersten and G. C. Murphy. Using task context to improve programmer productivity. In *Proc. of SIGSOFT '06/FSE-14*, pages 1–11, 2006.

[7] D. Khushraj, O. Lassila, and T. Finin. sTuples: semantic tuple spaces. *Mobile and Ubiquitous Systems: Networking and Services*, 2004.

[8] A. Ko, H. Aung, and B. Myers. Eliciting design requirements for maintenance-oriented IDEs: a detailed study of corrective and perfective maintenance tasks. In *Proc. of ICSE '05*, pages 126–135, 2005.

[9] A. Ko, R. DeLine, and G. Venolia. Information needs in collocated software development teams. In *Proc. of ICSE '07*, pages 344–353, 2007.

[10] O. Lassila and R. R. Swick. Resource description framework (RDF) model and syntax specification. `http://www.w3.org/TR/1999/REC-rdf-syntax-19990222/`.

[11] T. LaToza, G. Venolia, and R. DeLine. Maintaining mental models: a study of developer work habits. In *Proc. of ICSE '06*, pages 492–501, 2006.

[12] B. McBride. Jena: a semantic web toolkit. *Internet Computing*, 2002.

[13] L. Nixon, O. Antonechko, and R. Tolksdorf. Towards semantic tuplespace computing: the semantic web spaces system. In *Proc. of SAC '07*, pages 360–365, 2007.

[14] E. Prud'hommeaux and A. Seaborne. SPARQL query language for RDF. `http://www.w3.org/TR/rdf-sparql-query/`.

[15] T. Smith and M. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147:195–197, 1981.

[16] J. Stylos and B. Myers. Mica: A web-search tool for finding API components and examples. *Proc. of VL/HCC '06*, pages 195 – 202, 2006.

[17] D. Čubranić and G. Murphy. Hipikat: recommending pertinent software development artifacts. In *Proc. of ICSE '03*, pages 408–418, 2003.

[18] D. Čubranić, G. Murphy, J. Singer, and K. Booth. Hipikat: a project memory for software development. *Software Engineering*, pages 446–465, 2005.

[19] M. Wicks and R. Dewar. Ad hoc tool integration; a case study. *The International Conference on Software Development*, 2005.

[20] Y. Yang and J. Han. Classification of and experimentation on tool interfacing in software development environments. *Proc. of Asia-Pacific Software Engineering Conf.*, 1996.

List of Figures

List of Tables

Page Type	Percent	Count
Searches		
Code elements	12	76
Code ideas	11	72
Tools	5	32
Concepts	3	20
Sites	3	18
	34 %	218
Documentation		
Official	30	191
Unofficial	3	19
	33 %	210
Other resources		
Content	24	157
Project	5	33
Navigation	4	28
	34 %	218

Table 1: Taxonomy of development-related web pages. Percentages are of the total number of development-related pages, and do not sum to 100 due to rounding error.

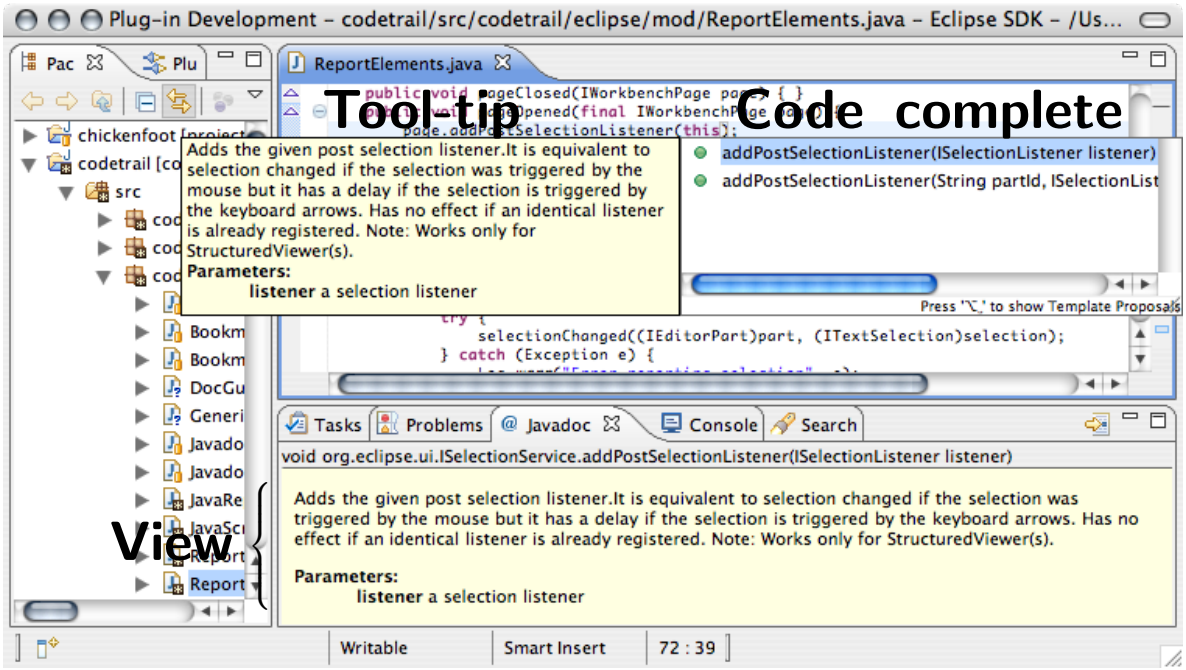

Figure 1: Built-in Eclipse features for accessing Javadocs: *tool-tips* and *code completion* suggestions presented within code, and a *view* of documentation alongside it.

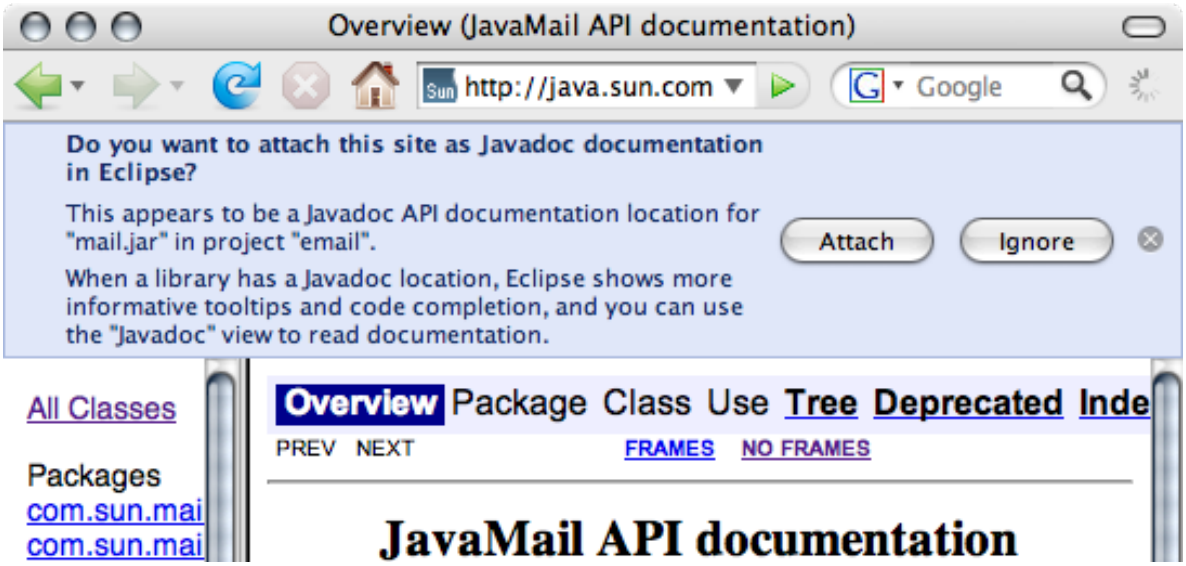

Figure 2: Javadoc attachment prompt shown when the user browses to a Javadoc documentation source that may be linked to a project library in Eclipse.

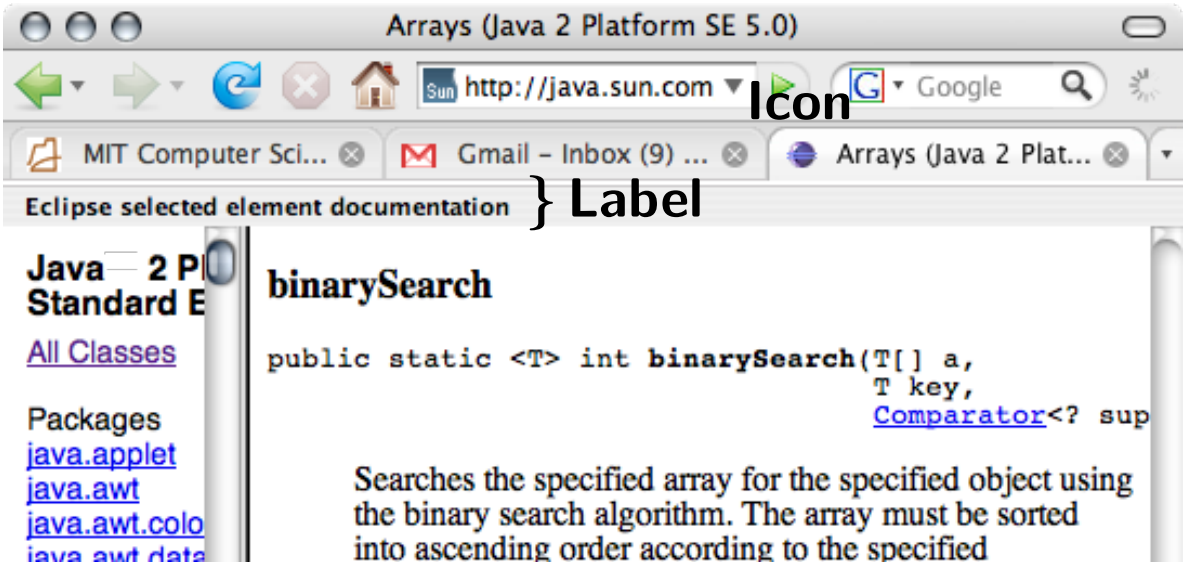

Figure 3: Automatic browsing tab, containing documentation automatically retrieved by Codetrail, with an identifying *icon* and *label*.

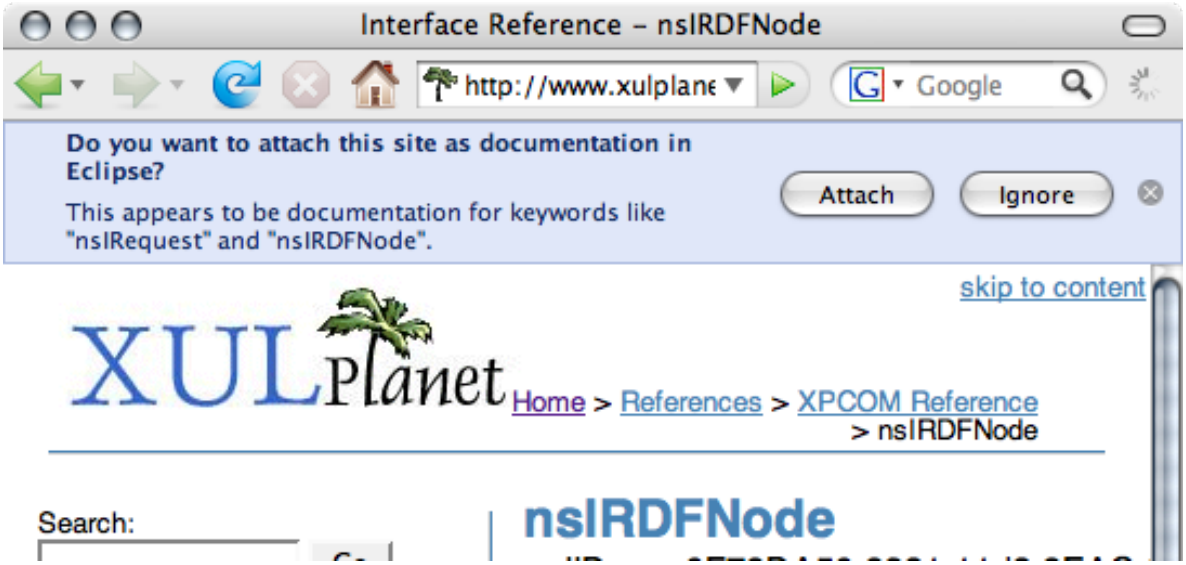

Figure 4: Generic documentation prompt shown when the user browses within a site that may be linked to keywords in project files.

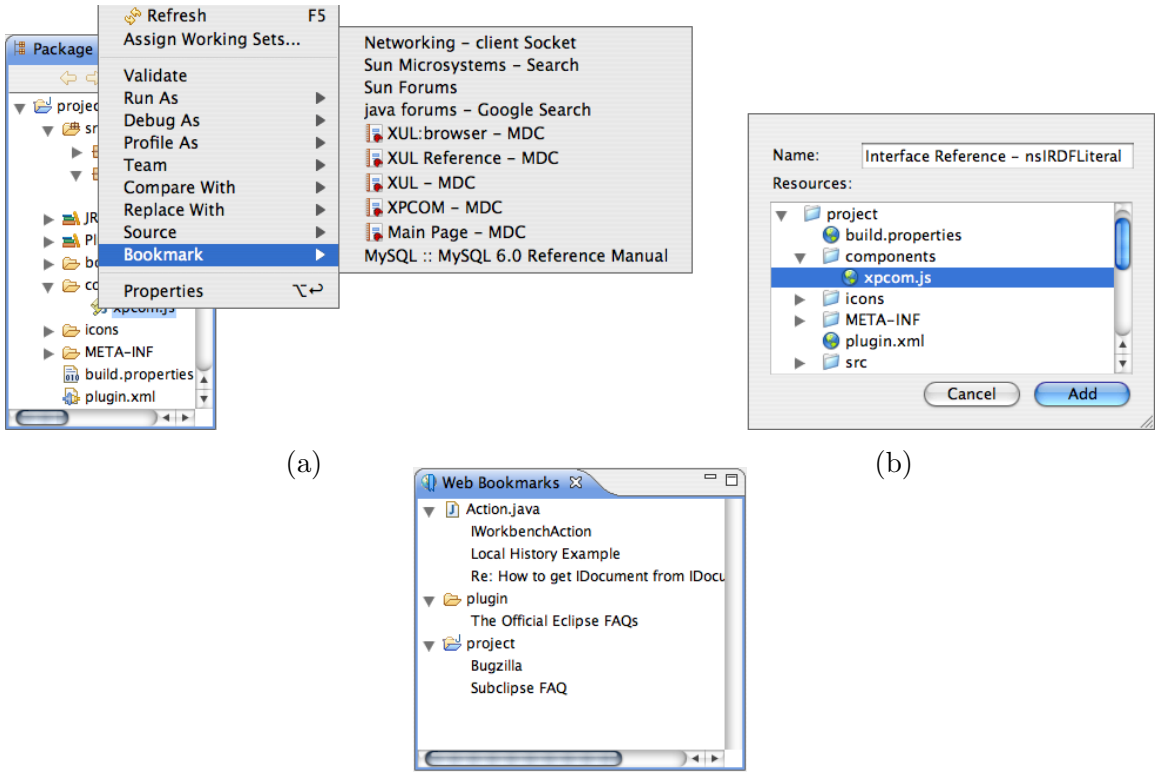

Figure 5: (a) Eclipse and (b) Firefox bookmark creation interfaces and (c) Eclipse bookmark view, showing bookmarks attached to an artifact and its containers.

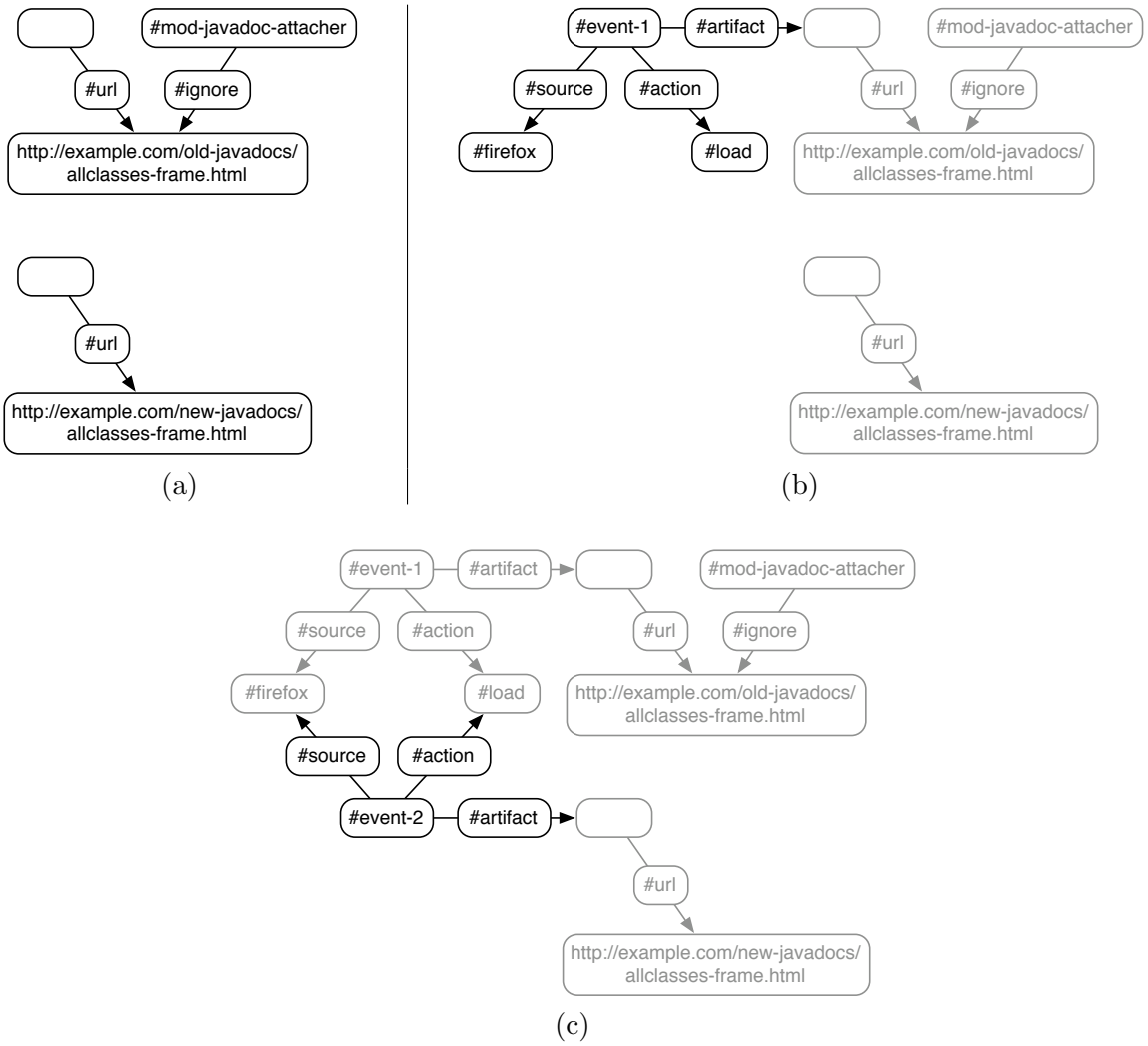

Figure 6: The shared RDF graph (a) before browsing, (b) after browsing to an ignored Javadoc site, and (c) after browsing a new Javadoc site.

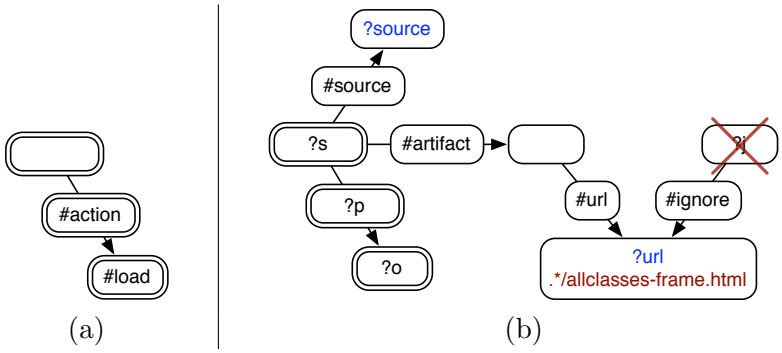

Figure 7: Example (a) template triple and (b) anchored query used by the automatic Javadoc identification module to observe visits to potential sources of Javadoc documentation.