Hipikat: Recommending Pertinent Software Development Artifacts

Davor Čubranić and Gail C. Murphy

Department of Computer Science
University of British Columbia
201-2366 Main Mall, Vancouver BC
Canada V6T 1Z4
E-mail: {cubranic, murphy}@cs.ubc.ca

Abstract

A newcomer to a software project must typically come up-to-speed on a large, varied amount of information about the project before becoming productive. Assimilating this information in the open-source context is difficult because a newcomer cannot rely on the mentoring approach that is commonly used in traditional software developments. To help a newcomer to an open-source project become productive faster, we propose Hipikat, a tool that forms an implicit group memory from the information stored in a project's archives, and that recommends artifacts from the archives that are relevant to a task that a newcomer is trying to perform. To investigate this approach, we have instantiated the Hipikat tool for the Eclipse open-source project. In this paper, we describe the Hipikat tool, we report on a qualitative study conducted with a Hipikat mock-up on a medium-sized in-house project, and we report on a case study in which Hipikat recommendations were evaluated for a task on Eclipse.

1. Introduction

A software developer who joins an existing software development team must come up-to-speed on a large, varied amount of information before becoming productive. Sim and Holt, for instance, interviewed newcomers to a project and found that they had to learn intricacies about the system, development processes being used, and the organizational structure surrounding the project, amongst others [15]. In collocated teams, this knowledge is often gained through mentoring: An existing member of the team works closely with the newcomer, looking over their shoulder, and imparting the oral tradition of the project, as the newcomer works on their first assigned tasks [15, 6].

Newcomers to open-source software projects cannot rely on this traditional approach. Communication between members of a team who are separated geographically and across time zones do not have access to such lightweight, high-bandwidth communication channels as the lunch table. Instead, these newcomers must try to find their way amongst the huge amount of archived, electronic information that is maintained as part of the project, such as the source repository, the bug database, and postings to newsgroups. Finding one's way, for instance, through the over 50,000 e-mail messages archived for the Apache open-source project (covering the period from February 1995 to May 1999 [11]) or through the one hundred or so postings *each* weekday to the Eclipse open-source project newsgroup,[1] is indeed a daunting task.

To help a newcomer in this situation become productive more quickly, we propose a tool that *recommends* existing artifacts from the development that are relevant to a task that the newcomer is trying to perform. In essence, we consider all of the artifacts that have been produced—the versions of the source, the bugs, archived electronic communication, web documents—as an implicit group memory. The tool plays two roles. First, the tool infers links between the artifacts that may have been apparent at one time to members of the development team but that were not recorded. Second, using these links, the tool, in a role similar to that of a mentor, suggests possibly relevant parts of the group memory given information about a task a newcomer is trying to perform. To investigate this hypothesis, we have built a tool called Hipikat,[2] we have instantiated Hipikat for the Eclipse open-source project, and we have conducted two exploratory studies: one using a mock-up of Hipikat on a medium-sized in-house project, and one using the Hipikat prototype on Eclipse.

In this paper, we begin with an overview of the structure of open-source projects, with a specific focus on Eclipse (Section 2). We then describe our approach to forming a

[1] www.eclipse.org

[2] Hipikat means "eyes wide open" in the West African language Wolof.

Proceedings of the 25th International Conference on Software Engineering (ICSE'03)
0270-5257/03 $17.00 © 2003 IEEE

group memory (Section 3), describe the Hipikat tool (Section 4), and present our exploratory studies into the effectiveness of the approach (Section 5). A discussion of the approach and tool follows (Section 6), as does a comparison to related efforts (Section 7).

2. Open-Source Projects

Outside the basic principle that the source code be freely available to anyone wishing to view or modify it [12], open-source projects vary in size, application domain, implementation language, team organization, and licensing model. Although there are differences between projects, a set of open-source best-practices is emerging, and a non-trivial project will typically produce the following four electronic artifacts.

1. *CVS source repository* Most projects use CVS [2] to provide version control support. The repository contains a complete history of the code base.

2. *Issue-tracking system* An issue-tracking system is typically used to submit and track bugs and feature requests on-line. Some systems, such as Bugzilla,[3] allow the posting of comments within tracked items, enabling the capture of discussion about the direction of bug fixes and feature implementations.

3. *Communication channels* Since project members are typically widely distributed, communication tends to be asynchronous, and to be supported electronically [5]. The culture of open-source projects is to keep all communication public; mailing lists and newsgroups are used more frequently than person-to-person mail. These archives are typically available on-line.

4. *Online documentation* Available documentation is most often code-oriented: reference manuals and programming guides are common; architectural descriptions and design-level documentation are rare. Documentation is usually distributed through a web site.

2.1. Eclipse.org

We are focusing our research efforts on one particular open-source project, the Eclipse extensible integrated development environment project. The Eclipse platform was originally developed by IBM, and was subsequently released under an open-source license. The platform can be extended through *plug-ins*. Basic Eclipse distribution includes plug-ins for Java development and for communication with CVS.

[3] `www.mozilla.org/projects/bugzilla/`

The Eclipse project mostly matches the earlier description of open-source projects: it includes a CVS repository with open read-only access, public newsgroups and mailing lists, a web site with documentation, and uses Bugzilla. At this time, there are two fairly distinct groups of developers: IBM employees who are working on the runtime of the platform, and on bundled plug-ins (the Eclipse *core*), and a much larger community of third-party plug-in developers, who range from large commercial software developers to hobbyist programmers. This division is reflected in the project's artifacts. For example, most newsgroups postings are questions about using and accessing the core API. Compared to the Mozilla project,[4] there is little discussion or give-and-take within the issue-tracking system, likely because the Eclipse project is in its early stages compared to Mozilla, which after four years as an open-source project has a sizeable portion of developers outside the Netscape Communications Corporation in which the system originated.

3. Approach

Our approach has two parts. First, we form the implicit group memory from the artifacts and communications stored in a project's history. Second, we present to the developer artifacts selected from this implicit group memory that may be relevant to the task being performed.

Figure 1 shows the schema we use to represent a project's implicit group memory. There are four types of artifacts represented in the schema: bug and feature descriptions (e.g., items in Bugzilla, source file revisions (e.g, checked in a CVS source repository), messages posted on developer forums (e.g., newsgroups and mailing lists), and other project documents (e.g., design documents posted on the project's web site). These artifacts are created by project members, represented by `Person` in the diagram.

Entries in the project's issue-tracking system are a locus within the schema because these entries typically represent a logical unit of work on the project. Source revisions are checked into the source repository to respond to one of these entries; newsgroup postings and mailing list messages often contain discussion that either results in a new entry or that is about an existing entry; other documentation may contain information about a particular entry, such as specific design trade-offs related to a feature request.

A given project may not contain all artifacts or links described in the schema. In general, missing *artifacts* result in a loss of the portion of the schema related to the artifact. In a small number of cases, such as structural design documentation, a missing artifact might be reverse engineered from other project artifacts; for example, some limited kinds of

[4] `www.mozilla.org`

Proceedings of the 25th International Conference on Software Engineering (ICSE'03)
0270-5257/03 $17.00 © 2003 IEEE

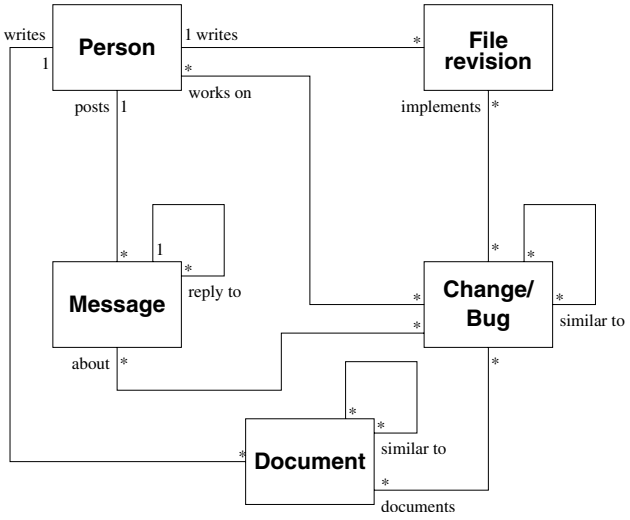

Figure 1. Artifact linkages schema

design documentation might be regenerated from source. Missing *links* are easier to infer as information contained within the project artifacts, and meta-information available about those artifacts, can be exploited. For instance, some links between feature requests and file revisions might be inferred if there is a project convention to include in a check-in comment associated with the revision a reference to the issue-tracking system entry that describes the feature request. Other links between entries in the issue-tracking system and file revisions might might be inferred based on meta-information, such as when particular project artifact's were created or touched; for example, it is likely that the author of a bug fix checked in source revision(s) close to the time that the bug was closed in the issue-tracking system.

The schema is also used to direct the selection of relevant artifacts in response to a query. For example, once a developer has started working on a task, such as a request to make a particular change to the system, the developer may be interested in other tasks that have been completed within the same subsystem, or with a similar description. Following the `similar_to` links may lead to tasks that are helpful. Once a similar task has been identified, following the `implements` links will lead to source revisions that implemented the task of interest. These revisions may help a developer identify code that may have to be modified or understood for the task at hand. The completed similar tasks may also have related discussions about which design options were examined, and which decisions were made that may impact the task at hand.

Abstractly, a tool implementing our approach has to implement three distinct functions.

1. **Identification** As artifacts are added to a project's history, the implicit group memory must be formed, including inference of missing links and artifacts.
2. **Selection** In response to queries, relevant artifacts must be identified and returned.
3. **Update** The project's archives must be monitored for additions and changes that result from the development and evolution of the system. The implicit group memory must be updated to reflect the additions and changes.

4. Hipikat Prototype

The Hipikat prototype is a client-server system. Hipikat is instantiated currently for the Eclipse project, but has been designed to be adapted easily to other open-source projects that follow the general model described in Section 2.

The client, when commanded by the user, issues a request for suggestions to the server, and displays returned results to the user. There are three parameters in any request from the client for suggestions. Two of the arguments are required: the first identifies anonymously the user,[5] and the second identifies artifact for which related items are sought. An optional third argument is intended to further describe the context of the query for additional tailoring of recommendations, although it is not used at this time. The server replies with a list of matches that the client then formats and presents in human-readable format.

4.1. Hipikat Client

Since Eclipse is self-hosted, we wrote the client as a plug-in that works within the IDE. This approach permits the Hipikat client to integrate seamlessly into a full-featured work environment, and to thus be used in combination with other software engineering tools plugged into Eclipse. For example, an Eclipse developer can use both Hipikat and the Java search feature that comes bundled with the default Eclipse distribution.

A developer who wants to make a query to Hipikat selects an artifact in their Eclipse project workspace, such as a class in a Java package browser, and chooses "Query Hipikat" from a pop-up context menu. (See Figure 2 for a full list of places in the IDE where such queries can be made.) Identifier of the selected artifact is passed as the second argument in the request to the Hipikat server, described in Section 4.2.

Additionally, the Hipikat artifact database can be searched based on search terms specified by the developer. This functionality is accessed through a pane in the regular

[5] Users are represented in the query to facilitate future extensions to selection mechanisms such as user-modelling and collaborative filtering. In the interests of privacy, user ids used in queries do not personally identify the user.

Proceedings of the 25th International Conference on Software Engineering (ICSE'03)
0270-5257/03 $17.00 © 2003 IEEE

Bug report open in the Bugzilla editor
CVS-managed file open in the Java editor
File in the CVS Repository view
Revision in the CVS Resource History view
CVS-managed file in the workspace Navigator
Item recommended in the Hipikat Results view
Bugzilla search match in the Search results view
Java class or method in Outline and Hierarchy views

Figure 2. Places where Hipikat query can be done in Eclipse

Eclipse search dialog, which also supports plain-text and Java-specific searches.

The results of a query or search are displayed in a Hipikat view within Eclipse (see Figure 5). The recommendations are grouped by artifact type and by selection criteria as determined by the identification submodule that reported a link. A recommendation can be opened for viewing, either within Eclipse or through an external viewer (e.g., a Web browser) depending on the artifact. We also provide access to Eclipse's revision comparison functionality to allow a developer to view a (CVS) artifact's differences from the preceding revision; for instance, to allow a developer to focus on particular changes to the source base. A Hipikat user can reorganize the recommendation lists, delete unwanted items, or move items towards the top or bottom of the list. Currently, this feature is purely for a user's direct benefit and convenience; in the future, this feature is intended be used to rate recommended items so that collaborative filtering capabilities can be added. Lastly, any recommendation can be used in another Hipikat query to help support easy traversal of the links in the group memory.

4.2. Hipikat Server

The server implements the three functions described earlier: identification, update, and selection. As Figure 3 shows, each function is encapsulated in a module. Each module is divided into submodules based on an artifact or link type for which it is responsible.

Update The update module for Eclipse has four submodules: one each for Bugzilla, CVS, newsgroups, and the Web site. Each update submodule monitors the project artifacts as appropriate for its type: The Web site is scanned by crawling the links, the CVS repository is monitored by parsing the history of updated files as gathered via `cvs update` commands, news articles are downloaded using the NNTP protocol from the Eclipse news server, and issue items are parsed from the output of Web front-end to Bugzilla.[6] New and changed artifacts are inserted into Hipikat's artifact database, and change listeners in the identification module are notified of the updates.

The artifact database saves primarily the *metadata* from the new and changed artifacts that are needed to establish relationships between the artifacts. For instance, for a CVS revision, the author and check-in time will be retained. Text, such as the bug descriptions or check-in comments, is indexed, both for the purpose of searching and for making similarity comparisons, but otherwise artifacts contents (i.e., the actual source of a Java file under CVS) that are not used to make recommendations are not kept.

Identification The identification submodules register with the update module as listeners for changes on artifact types for which they are responsible. There are currently four such submodules: check-in comment matcher (*log-matcher*), check-in time matcher (*activity-matcher*), text similarity matcher, and newsgroup thread matcher. When informed of a new instance of an artifact, or a change to an existing artifact, the identification submodules attempt to infer links within the implicit group memory, following the schema from Figure 1.

The *log-matcher* and the *activity-matcher* infer the `implements` links between a source revision and an issue ticket based on conventions used by Eclipse project developers. The *log-matcher* responds to changes in the CVS database, scanning CVS check-in comments for text that looks like bug id's (e.g., "Fixes bug *id*" or "*id*: ..."). The *activity-matcher* listens to updates to Bugzilla, taking advantage of the fact that Bugzilla is increasingly used to keep track of *all* the work done in the project, not just bugs reported by users. Since a developers will usually post a message on Bugzilla (and change the issue ticket's status) shortly after relevant source changes have been checked in, the *activity matcher* looks for checkins that are close to (within six hours of) activity occurring in a Bugzilla item. Checkins are further grouped into likely work units by looking for all checkins by a given developer within a small time window, similar to the strategy employed by Mockus, Fielding, and Herbsleb in their study of Mozilla development process [11].

The text similarity matcher works in two phases. First the text of new artifacts is indexed and each artifact—for example, a Bugzilla description and its comments, or a document on the project Web site—is turned into a *document vector* whose dimensions correspond to words in the vocabulary. The component magnitudes are the weights of words in the document and are calculated using a product of the term's *global weight*, indicating its overall importance in the entire collection, and *local weight*, which

[6] A production system could interface directly with the Bugzilla SQL database.

Proceedings of the 25th International Conference on Software Engineering (ICSE'03)
0270-5257/03 $17.00 © 2003 IEEE

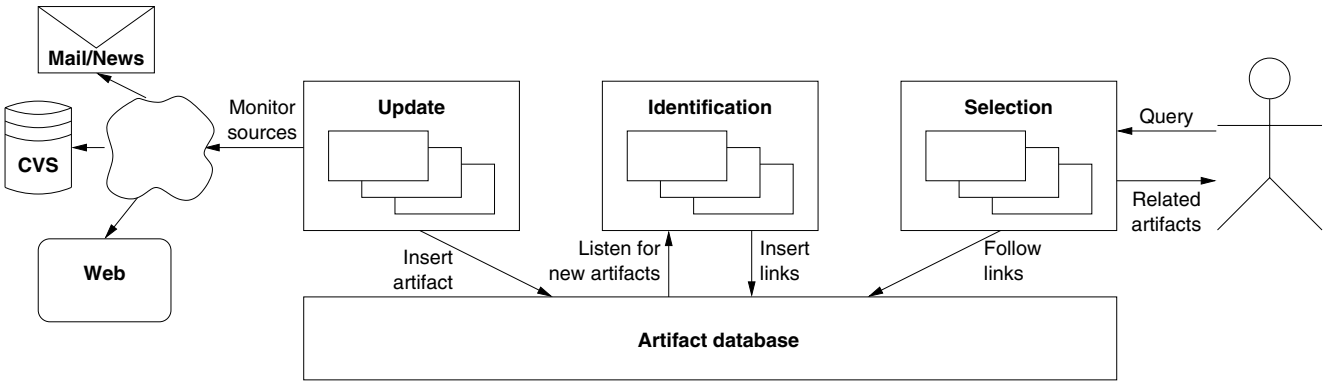

Figure 3. Hipikat server architecture

depends only on the frequency of terms within each document. We use log-entropy[7] combination for the two weights: The local weight is calculated as $L(i,j) = log(1+tf_{ij})$ and the global weight is calculated as $G(i) = 1 - \frac{1}{log(N)} \sum_{j=1}^{N} p_{ij} log(p_{ij})$, where tf_{ij} is the number of times term i occurs in document j, N is the number of documents in the collection, $p_{ij} = \frac{tf_{ij}}{d_i}$, and d_i is the number of documents containing term i.

In the second phase, the text similarity matcher uses a standard information retrieval vector-space cosine similarity measure [14] to infer `is_similar_to` links between artifacts. Specifically, the similarity between two documents is calculated as $sim(d_i, d_j) = \frac{d_i \cdot d_j}{\|d_i\| \|d_j\|}$. A selection submodule is responsible for using the computed measures to recommend a small set of nearest neighbours to an artifact.

This text similarity approach is also used in user-specified search queries: A user's query is treated just as another document vector, allowing matching artifacts to be sorted by relevance based on their degree of similarity to the search query.

Finally, the simplest identification submodule is the newsgropu thread matcher, which looks for "References" headers in newsgroup articles and reconstructs conversation threads of a newsgroup posting and subsequent replies.

Selection Lastly, selection works by following links from the artifact specified in client's request. Submodules are specialized to make recommendations for a subset of artifact types and their links—for example, one module makes recommendations on CVS and Bugzilla artifacts by following `implements` (and its reverse, `is_implemented_by`) links. Recommendations from all selection submodules to a given query are merged together before a final list is returned to the user.

Implementation The server is written as a Web application running within Tomcat.[7] The server and client communicate using SOAP [3].

To make a query, the client makes a remote procedure call, `getRecommendations`, with the three arguments described earlier: the anonymous user making the query, the id of an artifact for which recommendations are sought, and the currently unused third argument to provide additional context for making recommendations. The id for an artifact includes both a type (for quick selection of appropriate modules to handle the request), and a further unique identifier within the artifact type name space, as appropriate to the type (e.g., file path and revision number for a CVS artifact).

The server replies with an XML-formatted list of matches which is modelled on the "What's Related" service provided by Alexa.com and available in all major browsers. The recommendations are wrapped in a `RecommendationList` XML element (Figure 4). Each item recommended by the server is represented as a `Recommendation` element, with children `key`, `name`, `reason`, and `confidence`. The `key` element uniquely identifies the recommended artifact and is used if the user decides to open it or make a subsequent query on it. The `name` element is a human-readable description of the artifact and depending on its type can be a CVS file name, a one-line summary of a Bugzilla item, subject and author of a newsgroup posting, etc. The `reason` element describes why the item was recommended, and `confidence` expresses the relative strength of this relationship. The confidence value can be descriptive, as in "Same discussion thread" for a newsgroup posting, or numeric in the case of a text similarity measure.

The Eclipse project produces all of the kinds of artifacts in our schema. As of September 2002, our Hipikat server knows about 21,668 Bugzilla items (incorporating 72,536 additional comments), 125,429 CVS revisions,

[7] `jakarta.apache.org/tomcat`

Proceedings of the 25th International Conference on Software Engineering (ICSE'03)
0270-5257/03 $17.00 © 2003 IEEE

```
<RecommendationList>
  <Recommendation>
    <key>cvs:dev.eclipse.org:/home/eclipse/...
        org.eclipse.team.cvs.ui/src/org/eclipse/team/...
        internal/ccvs/ui/actions/TagAction.java:1.13</key>
    <name>org.eclipse.team.cvs.ui/src/org/eclipse/team/...
        internal/ccvs/ui/actions/TagAction.java:1.13</name>
    <reason>Bug ID in revision log</reason>
    <confidence>High</confidence>
  </Recommendation>
  <Recommendation>
    <key>bugzilla:12367</key>
    <name>[CVS Repo View] ``Define Branch Tag''...
        confusing?</name>
    <reason>Text similarity</reason>
    <confidence>0.6240631</confidence>
  </Recommendation>
  ...
</RecommendationList>
```

Figure 4. A sample response from Hipikat server.

36,864 newsgroup postings, and 1,459 Web pages (including those used solely to group frames and navigation functions) associated with the Eclipse project.

5. Validation

5.1. Initial Qualitative Study

Our first investigation of the Hipikat approach focused on whether recommendations of relevant artifacts were of any help to developers working on a change task, and if so, which kinds of recommendations were used. We also wanted to determine if there were recommendations developers would have found useful but that Hipikat did not suggest. For this initial assessment, we chose to manually build the implicit group memory for a medium-sized software system that was under development in our research lab. We then trialled a mock-up of Hipikat for this implicit group memory as part of an assignment for a graduate software engineering class held in our department.

The medium-sized software system used was the AVID visualization tool, which allows a developer to analyze the execution of a software system off-line in terms of an architectural view of the system [17]. AVID is written in Java and comprises 12,853 non-comment, non-blank lines of code organized in 177 classes and 16 packages.

Since the AVID development did not use a bug-tracking system, we analyzed its CVS repository, extracted about two dozen distinct change tasks from its history, including both bug fixes and new functionality, and entered these tasks as items in a Bugzilla database. We then built up the implicit group memory for the project, which consisted of change tasks, CVS revisions, documentation, project-related emails, and links between these artifacts. Although we formed the links manually, we followed the principles described in Section 3: revisions were linked to change tasks based on similarity of the check-in comments to task descriptions, documentation was associated through text similarity, and associations between change tasks were inferred using text similarity and overlap in source files that were changed as part of a task. These links were realistic: They contained both relevant and irrelevant suggestions since they were based on the information recorded. For instance, change tasks that contained similar words in their description may have been about something totally different, and groups of CVS checkins often included some revisions that were not related to the check-in comment or the rest of the group.

The mock-up client was a stand-alone Java application that allowed the user to browse the CVS repository the online documentation, and change task database. When the client displayed an artifact, it would issue a background request to the Hipikat server for items related to the displayed artifact, and would display a list of related artifacts returned in a side pane. The client also included a "bookshelf" where users could create notes as they worked and keep links to artifacts they accessed often.

The assignment in the graduate software evolution class consisted of having students implement two changes to AVID that we had selected from previously completed changes on a development branch of the system. Students were grouped into pairs; each pair was asked to use the mock-up for one change and one of Rigi[8], Chava[9], or jRMTool[10] for the other. We randomized the assignment of tools to the changes, and we randomized the order in which we asked the pairs to make the changes. Following the completion of the assignment, we invited students to participate in the study by sharing their assignment reports with us. Seven pairs (out of twelve in the class) agreed and gave us a copy of their assignment reports. We analyzed the comments in the reports and interviewed six of the subjects.

Overall, the subjects reported that Hipikat helped them to start the assigned task. In particular, suggestions of relevant previous changes to AVID that were based on textual similarity to the change at hand helped to identify the classes and methods that the subject needed to understand or modify to complete the assigned task:

> The suggestions on the side pane on the left gave us the starting point of classes to look for. We then used a bottom-up approach—browsed through the source code to see whether it is relevant to the change task. (Pair 5)

One potential problem with our approach is that a new-

[8] www.rigi.csc.uvic.ca
[9] www.research.att.com/sw/tools/chava/
[10] www.cs.ubc.ca/\~murphy/jRMTool/doc/index.html

Proceedings of the 25th International Conference on Software Engineering (ICSE'03)
0270-5257/03 $17.00 © 2003 IEEE

comer receiving suggestions may not have sufficient knowledge of the system to determine readily what a suggestion means and whether it is relevant. For the most part, the subjects in our study were able to distinguish which suggestions were likely relevant, and which were the result of apparent similarity However, some pairs reported difficulty in assessing relevance without "wasting a lot of effort" (Pair 5) investigating a suggestion. We also had reports of a pair missing a relevant suggestion because they lacked knowledge about the overall structure of the system, and realizing its relevance only once they have figured out the solution on their own.

> For the implementation of the change we ignored the existance [sic] of the AbstractQueryManager although the Hipikat tool more or less directly pointed us to it through the change task for [. . .] (Pair 7)

The usefulness of suggestions made by Hipikat depends on the context in which a suggestion is made and on the experience of the developer(s) receiving a suggestion as can be seen by the following two comments which talk about the same change task and the same set of recommendations:

> Unfortunately, none of the change tasks previously recorded in Hipikat bore much resemblance to the change we were attempting aside from identifying a file. . . (Pair 2)

and

> Hipikat definitely helped us during the first part of the task. . . . We quickly found a related task [. . .] We confirmed that this task was related by examining the CVS differences in files that Hipikat indicated were involved in the change. This meant we knew which methods we needed to look at. (Pair 4)

The results of this initial study showed that it was possible to make reasonable suggestions using the inferences we had posited, and that those suggestions were (sometimes) useful to developers. We proceeded to implement the Hipikat prototype for Eclipse, incorporating two enhancements from the initial study: we added the return of a reason for a recommendation to a user, and we changed the user interface from automatically suggesting related artifacts to making suggestions based on a query from the user.

5.2. Case Study: An Eclipse Change Task

As an initial step in the evaluation of our Hipikat prototype for Eclipse, we undertook a case study in which we used Hipikat to aid the performance of a change task on Eclipse. We selected a completed enhancement for Eclipse that was logged in Bugzilla, and created an implicit group memory for Eclipse based on the project artifacts that were in existence when that Bugzilla item was entered. The subject in this study was the first author on this paper. At the time of this case, he had experience with Eclipse plug-in development, but had not contributed to the core of the Eclipse project and was in a position similar to that of a newcomer to the Eclipse.org project. We chose a completed enhancement to enable us to compare our solution with that developed by the Eclipse team.

The enhancement request we selected deals with the behaviour of Eclipse's CVS repository browser. The browser displays the files and directories residing in the repository, and shows their existing branches and tagged versions. The repository supports basic operations such as checking out files (from either a trunk or version branch, as well as a tagged version), viewing a file's revision history, and creating new branches and tagged versions. However, when a new version is created through the browser, a manual refresh (potentially requiring the overhead of communication with the server) is necessary before the new version appears in the browser. The desired enhancement is to have the browser update automatically in this situation.[11] We describe the subject's use of Hipikat for the study as if it was currently occurring to provide the context of how the tool was used.

The subject starts off by opening the Bugzilla item describing the modification request describing the task at hand in Eclipse. The subject then makes a Hipikat query from the context menu of the viewer displaying the modification request. A list of similar items appears in the search result window, Since this is an unsolved request, the list includes only other Bugzilla items selected for their textual similarity to the change task at hand (Figure 5).

Near the top of the list is bug 11419, whose one-line description of "Automatically add version to repo[sitory] view when tagging", sounds similar to the current change task. A double-click opens it, and the subject determines that it describes functionality almost equivalent to the functionality that is desired. The difference is that bug 11419 describes functionality that was activated from the context menu for files in Eclipse workspace; the equivalent of the CVS "tag" command, as opposed to "rtag" in the CVS repository browser. Since bug 11419 is marked as "fixed", the functionality must have been added into Eclipse. The subject makes another Hipikat query to get files related to the completed enhancement, hoping to learn something from the fix (Figure 6).

At the top of the results list for this query are file revisions that Hipikat retrieved following the `imple-`

[11] Bug 116, accessible at URL: `bugs.eclipse.org/bugs/show_bug.cgi?id=116`.

Proceedings of the 25th International Conference on Software Engineering (ICSE'03)
0270-5257/03 $17.00 © 2003 IEEE

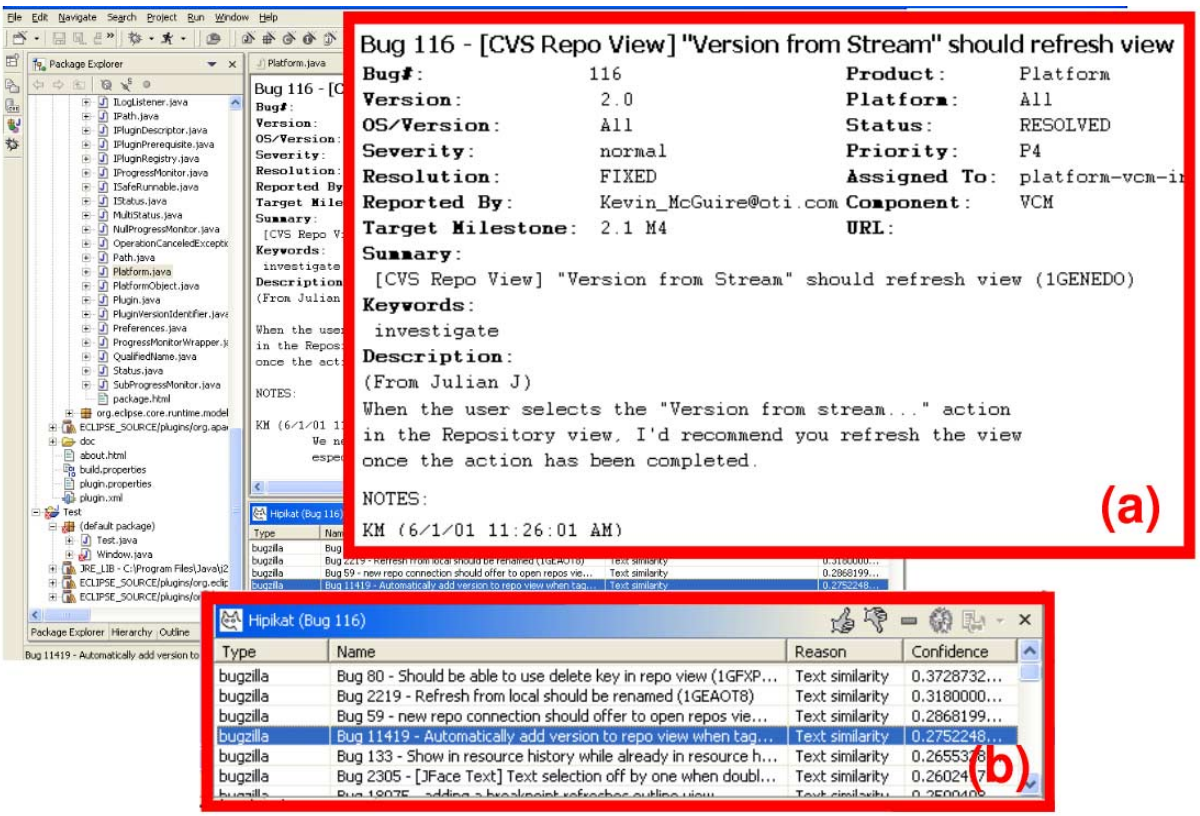

Figure 5. A screenshot of Hipikat being used within Eclipse IDE during the change task case study. Zoomed-in rectangles show the change task, bug 116 (a), and list of related artifacts (b)

`mented_by` links. In this case, there is a single file, *TagAction.java rev. 1.13*, which was recommended because the bug id 11419 appeared in its check-in comment, and because of the temporal proximity of its check-in to the closing of the item in the Bugzilla database. To focus on what it was that this revision changed in the code, the subject uses the context menu in the "Hipikat results" view to request a diff of the file to its preceding revision. The view comparing the revisions shows that the fix centers on a call to the CVS UI plugin's *RepositoryManager* from within `TagAction` class's `execute` method. The call is not entirely straightforward, because it also requires interaction with the model of CVS resources to find the tagged file's project in the workspace, and to map that file to the remote folder in the CVS repository.

At this point, the subject uses Eclipse built-in facilities to view the code of classes `TagAction` and `RepositoryManager`, as well as the classes in the model that are used in the `TagAction`'s code (e.g., `IResource`, `ICVSRemoteResource`, and `CVSTag`). With the subject's attention focused on just this part of the code base, it does not take long to figure out how tags are created in the model and updated in the view.

What remains is figuring out why the code works when called from the *Tag as Version* in the context menu on a file in the workspace, but not in the CVS Repository browser. Debugging the current version of `TagAction`, which includes the changes added in version 1.13, by, for instance, inserting a breakpoint, can help the subject identify quickly that it is not the action that is invoked for the CVS Repository browser's menu.

A look at the XML file that defines a plugin's contributions to various context menus shows that `TagAction` is executed only when the *Tag as Version* option is chosen from the context menu on items of type `IResource`, which are directories and files in the Eclipse workspace. Further on in the same file, the subject sees that for the context menu on items of type `ICVSRemoteResource`, `TagInRepositoryAction` is the action executed, so the subject starts to look at the code for that class.

Upon inspection of `TagInRepositoryAction`, the subject sees that its `execute` method is almost identical to the `execute` method in `TagAction` before changes implemented in the revision fixing bug 11419. This logi-

Proceedings of the 25th International Conference on Software Engineering (ICSE'03)
0270-5257/03 $17.00 © 2003 IEEE

Type	Name	Reason
cvs	...internal/ccvs/ui/actions/TagAction:1.10	Bug ID in revision log
bugzilla	Bug 12367 - [CVS Repo View] "Define Branch Tag" confusing?	Text similarity
bugzilla	Bug 8185 - Branch should automatically version	Text similarity
	...	

Figure 6. The top few artifacts related to bug 11419

cally leads the subject to apply the same approach in `TagInRepositoryAction`, which indeed solves the change task, as evidenced by the the fix implemented by Eclipse developers.

6. Discussion

Lessons from validation

For software change tasks, an inquiry episode has been defined as consisting of a developer asking a question, conjecturing an answer, and then searching through the code and documentation to verify or reject the conjecture [16]. Since the search space is so large, newcomers tend to have difficulty coming up not only with a good conjecture, but also the way of searching through the documentation and code to verify it. In our case studies, we found that Hipikat was most useful at the beginning of a major inquiry episode. A developer can query from a Bugzilla item describing the change, producing a set of similar bugs: Some of these bugs may serve as starting points for a newcomer to make more informed conjectures. Specifically, recommended fixed bugs will have associated source, which the developer can use to identify code possibly relevant to the current change.

Most of the time, the recommendations to bugs and source are still just entry points. The developer must still evaluate the code and understand how it works. For this part of the task, other source-based discovery tools, such as the Java search feature of Eclipse, can be used to help the developer focus on understanding a manageable (and relevant) portion of the system.

A more fundamental issue with Hipikat is that the quality and number of items recommended affects the newcomer's performance on the task. When the recommendation list is long, the newcomer can have trouble determining if any of the recommendations are relevant. In the study described in Section 5.1, we found that newcomers sometimes had difficulty assessing recommendations. Some newcomers who were recommended a fixed bug with many associated source files investigated all of associated files in sequence, even when there was little basis to believe a file was relevant for the current change. Reporting the amount of code involved in a revision when presenting the recommendations may alleviate this problem somewhat, although obviously no machine-based solution will work in all cases.

Eclipse We chose the Eclipse project as our testbed for two main reasons. First, Eclipse is a large system, comprising over 800,000 lines of Java code, for which a rich history of project artifacts is available. Second, we could integrate our tool into Eclipse, making it more accessible to Eclipse developers. Since Eclipse shares similar processes and structure with other open-source projects—for instance both Eclipse and Mozilla use Bugzilla to track all work done in a project—our approach may reasonably apply to other targets.

Accuracy Our link inference algorithms are approximate. Out of 9,418 bugs marked fixed as of August 24, 2002 in the Eclipse Bugzilla database, the Hipikat log- and activity-matcher identification submodules inferred revision links for 5,688 (60%) of these bugs. The two submodules linked a further 2,810 bugs to revisions even though these bugs were not marked fixed; most of these links are probably false, the result of bugs that have been reopened or that have been erroneously left open, or that refer to revisions implementing test cases for still-unsolved bugs. Over two thirds of these false links are ranked with lower confidence because they are for activity that is significantly distant from the time of the check-in: The confidence in a link inferred by the activity-matcher progressively decreases when this distance is over five minutes.

It remains to be seen how users will treat Hipikat recommendations in view of their approximativeness. Will a user give up on certain kinds of recommendations entirely once they conclude, rightly or wrongly, that they are not helpful, as happened during testing of our mock-up? Will Hipikat's "confidence" measures make sense? Since it has been shown that expressing confidence as a numeric value in recommenders makes little sense to users [8], we try to use textual descriptions wherever possible. In some places, like text similarity, there is no such obvious explanation and it remains to be seen whether users will make use of the numeric value (vector cosine), or if they will simply go through the list and view recommended artifacts, possibly filtering based on the short description of the artifact that is included in the recommendations view.

Beyond Search The Eclipse web site supports a search

Proceedings of the 25th International Conference on Software Engineering (ICSE'03)
0270-5257/03 $17.00 © 2003 IEEE

function. However, this search is of limited functionality, in part, because of the high amount of noise in its matches. As an example of the noise, the search includes the text of source files so that any query involving a class name will yield matches for *all* revisions of that class and other classes that use it. More fundamentally, searching is by definition useful when the user knows on what to search. Hipikat, on the other hand, allows a user who is working on a change task and who is unsure of where to start or what to look for to simply say "tell me about what might be similar to what I'm doing".

Role of Task We use Bugzilla entries as focal points to which other artifacts are linked. This organization makes sense in view of the fact that Bugzilla is increasingly used to keep track of *all* the work done in the project. The same process happened in Mozilla—which is now much further along this path, thanks to its longer duration and more diversified developer community—and we expect that as Eclipse evolves in the same way, all revisions implementing more than trivial changes will have corresponding Bugzilla items.

Furthermore, while Bugzilla is normally used with a Web browser front end, including viewing and editing of Bugzilla reports into Eclipse gives us a simple way of tracking what the developer's current change task is. Although we are not currently using it to refine the recommendations, having access to Bugzilla within the IDE lowers the barrier to use Hipikat on change tasks and makes it more likely it will be applied in the early stage of change task planning where it is most likely to be helpful.

Collaborative filtering and user modelling Currently, the identification of links between artifacts in Hipikat is purely content-based; it depends solely on the contents of artifacts, such as when artifacts are compared based on text similarity, and on metadata, such as in the bug activity matcher identification submodule. Collaborative-based techniques could be used to enhance link identification. For example, newsgroup postings from authoritative sources, such as developers who checked in revisions on the same subject, could be given higher priority when making recommendations.

Schema-based selection could also be further refined through the use of a user model. For example, the CodeBroker [18] reuse recommender keeps track of components in a reuse repository that a developer has often accessed in the past, assumes that these components are well-known, and thus does not include the components in its recommendations. Although collaborative filtering would, by definition, have to be done in the Hipikat server, user modeling could be done partly or even entirely by the client.

7. Related Work

Many existing approaches have attempted to use the artifacts associated with a software development, other than source, to aid in software evolution tasks. These approaches vary in their degree of automation and in their specificity to a task.

Several systems mine version control information. The VE editor can display the revision in which a given line was changed together with its check-in description, and supports browsing of the code through a sequence of versions [1]. The Expertise Recommender system (ER) uses the author information recorded in a program's change history to generate recommendations of people who might have some expertise on a given problem [10]. Neither of these tools attempts to integrate version information with other artifacts. For example, we have found in practice that a problem report often contains information beyond that in a version check-in description.

Initial steps towards integrating software artifacts with developers' communication were made by Lougher and Rodden [9], whose system allowed maintenance engineers to make annotations on the code, capturing rationale and making long-term collaboration possible. Their approach requires the user to look at the exact spot in the source code to see the annotation, which may not be as useful for a relative newcomer trying to grasp tens of thousands of lines of source. Anchored Conversations [4] coupled actual conversations between collaborators, either real time chats or asynchronous messages, with the work artifact for in-context discussion, but suffers from the same drawback when dealing with a multi-megabyte artifact corpus.

Similar to Hipikat, Ye and Fischer's CodeBroker [18] uses information retrieval methods to determine software artifacts to suggest in the context of a developer's current task. However, CodeBroker is tailored to helping a developer on small-scale reuse tasks: The tool monitors a developer's use of a text editor watching for the method declarations and the descriptions of those methods in comments, the tool uses that information as a query to a library to find potential components that could be reused instead of a new component being created. In contrast, when used as a reuse tool, Hipikat works at the granularity of a task, providing such information as documents describing how a component is to be used with other components. The CodeBroker approach also relies on a developer properly formatting documentation in the component being defined, and in the presence of properly formatted documentation in the components in the reuse library. Hipikat avoids placing any additional requirements on the developers, making use of information that is potentially more informal. In this regard, Hipikat is more similar to the *Remembrance Agent* [13], which used information sources, such as user's email fold-

Proceedings of the 25th International Conference on Software Engineering (ICSE'03)
0270-5257/03 $17.00 © 2003 IEEE

ers and text notes, to present documents relevant, or similar, to the one currently being edited.

Mockus, Fielding, and Herbsleb have described building logical work units from CVS revisions by looking at revisions that were checked in by a single author within a small time interval and that share the same comment [11]. We relax their technique by allowing different check-in comments. The advantage of relaxing this constraint is that we may be more inclusive of the revisions that are part of the same change. The disadvantage is the increased cognitive load place don the developer who has to decide if a recommendation based on this information is relevant. We do note in the recommendations a lower-level of confidence when this inference heuristic is used and the check-in comments do not match. We further extend Mockus et al's method by attempting to match the checkin with the activity in the issue-tracking system as an additional way to link the code revisions with the descriptions of tasks in which developers were engaged.

8. Summary

The archives of a software development project can serve as an implicit group memory about the design of the system, decisions made over the course of the development, "tricks" for solving bugs, amongst many other kinds of information. Although the archives are stored and often accessible through common search techniques, the information is not integrated into a group memory. In this paper, we have described Hipikat, a tool that forms an implicit group memory for a project by inferring links between stored artifacts, and that then recommends pertinent parts of the group memory (artifacts) to a developer working on a task. Through two qualitative studies, we have shown that this approach shows promise for helping a newcomer perform a task effectively on an unfamiliar system.

Acknowledgments

This research was funded, in part, by NSERC and IBM (Ottawa Software Lab), as part of the Consortium for Software Engineering Research. We would like to thank the students of CPSC 507 in the fall of 2001 for their participation in our initial qualitative study.

References

[1] D. L. Atkins. Version sensitive editing: Change history as a programming tool. In *System Configuration Management 98*, pages 146–157. Springer-Verlag, 1998.

[2] B. Berliner. CVS II: Parallelizing software development. In USENIX Association, editor, *Proceedings of the Winter 1990 USENIX Conference*, pages 341–352, Jan. 1990.

[3] D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. F. Nielsen, S. Thatte, and D. Winer. Simple object access protocol, `http://www.w3c.org/TR/SOAP`, 8 May 2000.

[4] E. F. Churchill, J. Trevor, S. Bly, L. Nelson, and D. Čubranić. Anchored Conversations: chatting in the context of a document. In *CHI 2000*, pages 454–461, 2000. ACM Press.

[5] D. Čubranić and K. S. Booth. Coordinating open-source software development. In *WETICE 1999*, pages 61–65, 1999. IEEE Computer Society Press.

[6] M. Cusumano and R. Selby. *Microsoft Secrets: How the World's Most Powerful Software Company Creates Technology, Shapes Markets, and Manages People*. The Free Press, 1995.

[7] S. Dumais. Improving the retrieval of information from external sources. *Behaviour Research Methods, Instrument, and Computers*, 23(2):229–236, 1991.

[8] J. L. Herlocker, J. A. Konstan, and J. Riedl. Explaining collaborative filtering recommendations. In *CSCW 2000*, pages 241–250, 2000.

[9] R. Lougher and T. Rodden. Supporting long term collaboration in software maintenance. In *COOCS=93*, pages 228–238, 1993.

[10] D. W. McDonald and M. S. Ackerman. Expertise Recommender: A flexible recommendation system and architecture. In *CSCW 2000*, pages 231–240, 2000. ACM

[11] A. Mockus, R. T. Fielding, and J. Herbsleb. Two case studies of open source software development: Apache and Mozilla. *ACM Transactions on Software Engineering and Methodology*, 11(3):1–38, July 2002.

[12] B. Perens. The open source definition. In C. DiBona, S. Ockman, and M. Stone, editors, *Open sources: voices from the open source revolution*. O'Reilly, 1999.

[13] B. J. Rhodes and T. Starner. Remembrance agent. In *PAAM '96*, pages 487–495, 1996.

[14] G. Salton and M. J. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, New York, 1983.

[15] S. E. Sim and R. C. Holt. The ramp-up problem in software projects: A case study of how software immigrants naturalize. In *ICSE 1998*, pages 361–370, 1998. IEEE Computer Society Press / ACM Press.

[16] M.-A. D. Storey, K. Wong, and H. A. Müller. How do program understanding tools affect how programmers understand programs? *Science of Computer Programming*, 36(2–3):183–207, Mar. 2000.

[17] R. J. Walker, G. C. Murphy, B. Freeman-Benson, D. Wright, D. Swanson, and J. Isaak. Visualizing dynamic software system information through high-level models. In *OOPSLA 1998*, pages 271–283, 1998. ACM Press.

[18] Y. Ye and G. Fischer. Information delivery in support of learning reusable software components on demand. In Y. Gil and D. B. Leake, editors, *IUI 2000*, pages 159–166, 2002. ACM Press.

Proceedings of the 25th International Conference on Software Engineering (ICSE'03)
0270-5257/03 $17.00 © 2003 IEEE