# Hipikat: A Project Memory for Software Development

Davor Čubranić, Gail C. Murphy, *Member*, *IEEE Computer Society*, Janice Singer, and Kellogg S. Booth

**Abstract**—Sociological and technical difficulties, such as a lack of informal encounters, can make it difficult for new members of noncollocated software development teams to learn from their more experienced colleagues. To address this situation, we have developed a tool, named Hipikat, that provides developers with efficient and effective access to the group memory for a software development project that is implicitly formed by all of the artifacts produced during the development. This *project memory* is built automatically with little or no change to existing work practices. After describing the Hipikat tool, we present two studies investigating Hipikat's usefulness in software modification tasks. One study evaluated the usefulness of Hipikat's recommendations on a sample of 20 modification tasks performed on the Eclipse Java IDE during the development of release 2.1 of the Eclipse software. We describe the study, present quantitative measures of Hipikat's performance, and describe in detail three cases that illustrate a range of issues that we have identified in the results. In the other study, we evaluated whether software developers who are new to a project can benefit from the artifacts that Hipikat recommends from the project memory. We describe the study, present qualitative observations, and suggest implications of using project memory as a learning aid for project newcomers.

**Index Terms**—Software development teams, project memory, software artifacts, recommender system, user studies.

## 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, the development processes being used, and the organizational structure surrounding the project, amongst others [1]. In collocated teams, this knowledge is often gained through mentoring: An existing member of the team works closely with each of the newcomers, looking over their shoulders, and imparting the oral tradition of the project, as the newcomers work on their first assigned tasks [1].

As Berlin observed in her study of interactions between newcomers and mentors [2], mentors use these exchanges to provide a rich array of information, often tangential to the newcomers' actual questions, but nonetheless crucial for their development into experts. Initially, this extra information includes basic concepts relevant to the problem domain and tips on using the tools effectively. Over time, the focus switches to the system's design rationale, goals, and trade-offs. Mentors emphasize hard-to-find information that is typically difficult for the newcomers to acquire on their own: the unwritten design choices, historical quirks, or the code's assumptions and interactions between different modules. Mentors also introduce the newcomers to useful information sources, including other teammates' areas of expertise and relevant documentation and its reliability.

Most of the time, a newcomer works independently. The mentor is not like a tutor who is there all of the time, but rather the mentor checks up on the newcomer, perhaps once per day, monitoring the newcomer's progress and providing feedback and advice. The mentor is the person the newcomer turns to for help when stuck. These interactions are typically informal and lightweight, such as a quick question asked over the cubicle divider or at the water cooler during chance encounters.

These lightweight interaction channels are not always available in *virtual teams*, where the members of the team are not collocated. Moreover, studies show that workers are less likely to help their noncollocated colleagues [3], making it even harder for a newcomer to come up-to-speed on a project in a virtual team.

Fortunately, the situation is not hopeless. By their nature, virtual teams work and communicate through electronic media, such as mailing lists, source code versioning systems, and systems for recording and tracking work on issues such as problem reports and requested features. Arguably, the collection of all such artifacts created in the course of development of a software system implicitly forms a group memory—a repository of information that a work group can use to benefit from its past experience to respond more effectively to present needs [4], [5]. We call this implicitly-formed group memory the *project memory*.

- *D. Čubranić is with the Department of Computer Science, University of Victoria, Box 3055 STN CSC, Victoria, BC V8W 3P6, Canada. E-mail: cubranic@cs.uvic.ca.*
- *G.C. Murphy and K.S. Booth are with the Department of Computer Science, University of British Columbia, 201–2366 Main Mall, Vancouver, BC V6T 1Z4, Canada. E-mail: {murphy, ksbooth}@cs.ubc.ca.*
- *J. Singer is with the Institute for Information Technology, National Research Council Canada, M-50, 1200 Montreal Road, Ottawa, ON K1A 0R6, Canada. E-mail: janice.singer@nrc-cnrc.gc.ca.*

*Manuscript received 22 Oct. 2004; revised 18 Jan. 2005; accepted 2 Feb. 2005; published online 29 June 2005.*
*Recommended for acceptance by A. Hassan and R. Holt.*
*For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSESI-0227-1004.*

0098-5589/05/$20.00 © 2005 IEEE Published by the IEEE Computer Society

However, this information is not easily accessible because of its sheer volume, the lack of tools to navigate the repositories effectively, and the difficulty of making connections between logically related items in disparate repositories. General search engines, such as Google,[1] are commonly used for this purpose, but limit the developer to searching for exact words in documents within a single collection (e.g., the Web or a user's email). The developer has to know the right terms to use in the search, and the search engine cannot take advantage of the properties of different artifact types and relationships between them.

We have built a tool called Hipikat[2] to provide developers efficient and effective access to the project memory. We currently focus on open-source software projects, a subset of virtual teams that typically have extensive public archives of all artifacts relevant to the project. Despite the availability of artifacts, newcomers to open-source software projects still have a particularly hard time coming up-to-speed and learning about the system.

Hipikat is intended to assist newcomers by recommending items from the project memory—source code, problem reports, newsgroup articles, etc.—that are relevant to their current task, in effect making it easier for them to "learn from the past" even when a mentor is not available. The project memory is built automatically and with little or no change to existing work practices. We believe that this low barrier to introducing our tool into a project is crucial for Hipikat to be accepted in the real world because it has been repeatedly found that CSCW systems that require significant changes to work practice, or that require users to constantly externalize and map their knowledge, ultimately fail [6].

This paper presents our approach to the Hipikat tool, its implementation, and empirical investigation of its performance. We begin with an overview of related work. We then briefly describe our project memory model and the implementation of the Hipikat tool. We continue by presenting the results of two empirical studies. In one study, we evaluated the quality of Hipikat's recommendations on a sample of software modification tasks drawn from a large open-source software project. In the other study (the "Eclipse newcomers"), we investigated how Hipikat was used by a group of newcomer developers under realistic conditions. (The order in which we present the studies is not chronological: The Eclipse newcomer study was done first but is described second, and is only summarized in this paper for completeness. Its full details are in an earlier paper [7].) We conclude with a discussion of research results and future research directions.

## 2 Related Work

### 2.1 Group/Organizational Memory

A group memory for software development teams was proposed by Terveen et al. [8]. Their system, *Design Assistant*, guided the developer through a sequence of design decisions and produced a recommendation on code structure and usage of APIs. However, unlike Hipikat, Design Assistant relied on human experts for building and maintaining the group memory. It also required changes to the development process to be effective, something we expressly wanted to avoid.

Berlin et al. [5] presented a system, called TeamInfo, that built a group memory from messages that had been sent to a special email address. These messages were categorized automatically, using preconfigured keyword patterns, or explicitly by users. Hipikat similarly collects developer communication into a group memory, with the important distinction that it monitors activity in a public forum and, so, does not require the developers to "remember" to include it on the "CC" list. Hipikat also goes beyond Berlin et al.'s approach by correlating information from multiple sources (e.g., the discussion about a bug and the code implementing the solution). Berlin et al. used a taxonomy of categories to organize the collection; Hipikat recommends relevant items on a case-by-case basis.

Initial steps toward integrating information sources with little extra overhead required from users were made by Lougher and Rodden [9], whose system allowed maintenance engineers to make annotations on the code. The annotations supported asynchronous communication about the maintenance changes, while at the same time capturing the discussions and decisions that were made and associating them with the source code. The drawback of this approach is that it requires the developer to look at the exact spot in the source code to see the annotation, which may not be useful for a relative newcomer trying to grasp tens of thousands of lines of source in a multimegabyte artifact corpus.

### 2.2 Recommender Systems

By its philosophy, Hipikat is closest to recommenders like *Remembrance Agent* [10] and *CodeBroker* [11]. Remembrance Agent integrates with productivity applications like e-mail readers and text editors, mining information sources such as a user's email folders, and text notes to present documents relevant to the one currently being edited. CodeBroker likewise monitors the source code file that a developer is editing, watches for method declarations and the descriptions of those methods in comments, and uses information retrieval methods to recommend software library components that can potentially be reused instead. In contrast, when used as a reuse tool, Hipikat can work on a higher level of abstraction, providing information such as documents describing how a component is to be used with other components.

Both Remembrance Agent and CodeBroker work more like recommenders/search engines within a single collection, whereas Hipikat helps integrate multiple information sources. As an example of the usefulness of integrating information, we have found that problem reports stored in an issue tracking database often contain more information than is recorded as part of a check-in for the associated source code that fixes the problem. Automatically correlating this information can provide the developer with more useful information in a single search. There are other important differences. Unlike Hipikat, Remembrance Agent does not use information about the structure of the documents or their metadata to make recommendations.

1. http://www.google.com.
2. Hipikat means "eyes wide open" in the West African language Wolof.

The CodeBroker approach relies on a developer properly formatting documentation in the component being defined, and on the presence of properly formatted documentation in the components in the reuse library. Hipikat avoids placing any additional requirements on the developers, but instead makes use of less formal information.

### 2.3 Programming from Examples

One of the important coding strategies used by both beginner [12] and expert programmers [13], [14] is to use existing code as a template while developing a solution to the task at hand. This kind of code reuse philosophy is particularly strong in the world of open-source software and is partly the reason behind the tongue-in-cheek expression "Use the source, Luke:"[3] The system's source code is, in effect, full of examples of how to access the API, handle errors, and implement various functionality. However, it is difficult for a newcomer to build an understanding of a large software system simply by reading its source code. It is equally difficult to find useful examples in the source code since finding (and recognizing) them requires a certain level of understanding of the code. Hipikat helps newcomers find those examples by recommending modifications from a project's history similar to the user's current task. This approach differs from the existing systems that provide examples, such as *Reuse View Matcher* [16] and *Explainer* [17], whose libraries of examples are created specifically for that purpose, rather than drawn from existing related code.

### 2.4 Mining Software Artifact Repositories to Aid Software Maintenance

While the information in software repositories can be mined to study the evolution of the software for the purpose of retrospective analysis (e.g., see [18]), it is also directly useful to a developer engaged in maintenance and evolution tasks. Grinter observed developers' use of configuration management tools and reported that the history of changes was commonly accessed to learn what previous developers did [19]. A number of approaches have attempted to make it easier for a developer to access this information. Most existing systems focus on a single repository type, usually a source revision system such as CVS. For example, CVSSearch by Chen et al. [20] works as a search engine, where a developer can type in search terms and receive in response fragments of source code associated with the check-in comments in which the search terms occurred. *Expertise Browser* by Mockus and Herbsleb [21] and *Expertise Recommender* by McDonald and Ackerman [22] use source change data to identify developers and groups with experience in given elements of a software project. Zimmerman et al. [23] and Ying et al. [24] applied data mining techniques to source repositories to find change patterns, which can be used to make recommendations about potentially relevant files to a developer working on a software change.

3. The expression is a pun on a line in the original *Star Wars* movie trilogy, "Use the force, Luke [Skywalker];" the acronym "UTSL" is often used instead [15]. A common corruption is "read the source Luke," or "RTSL," probably derived by analogy to "read the [fine] manual," another piece of advice commonly given to newcomers.

## 3 The Principles of the Hipikat Approach

The core idea of our approach is to recommend artifacts created as part of the development of a software system that may be of relevance to a developer working on software evolution tasks for that system. Hipikat can be viewed as a recommender system for software developers that draws its recommendations from a project's development history.

There are two distinct functions performed by Hipikat. First, the tool forms a project memory from artifacts that were created during a software development project. The artifacts are not limited to source code and documentation (for example requirements specifications), but include communications conducted through electronic media that are captured (email messages or discussion forum postings), bug reports, and test plans. Second, Hipikat recommends to a developer artifacts selected from the project memory that it considers to be relevant to the task being performed.

These two functions are implemented in modules that operate concurrently and independently. Recommendations can be made as soon as any part of the project memory is created. The formation of the project memory is an ongoing process: As new project information is created, the project's information sources—the code repository, the issue database, etc.—are monitored for additions and modifications, and the project memory is updated accordingly. Depending on the information source, the monitoring may be continuous or periodic, and once the project memory updates are committed, they can be included in recommendations to users.

### 3.1 Forming the Project Memory

The project memory consists of the project artifacts themselves and also of links between those artifacts indicating relationships. Thus, we can model the project memory as an entity-relationship diagram [25]. Both artifacts and relationships are typed. There are four types of artifacts in our model, corresponding to artifacts that are typically created in open-source software projects: *change tasks* (i.e., problem reports and feature request descriptions recorded in an issue tracking system such as Bugzilla[4]), source *file versions* (e.g., checked into a source repository such as CVS[5]), *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). Fig. 1 shows the schema of these artifacts in the project memory together with the relationships we establish between them. The figure also shows a fifth entity, *person*, which represents the author of an artifact. Each artifact is uniquely identified by an artifact key, so that it can be referred to in queries and recommendations. The artifacts in the figure are annotated with the names of tools that produced them in the implementation of project memory presented here (in italics); however, Hipikat is designed to relatively easily accommodate other tools producing similar information.

Relationships (links) between the artifacts are established either from existing information about artifacts that is

4. http://www.mozilla.org/projects/bugzilla.
5. http://www.cvshome.org.

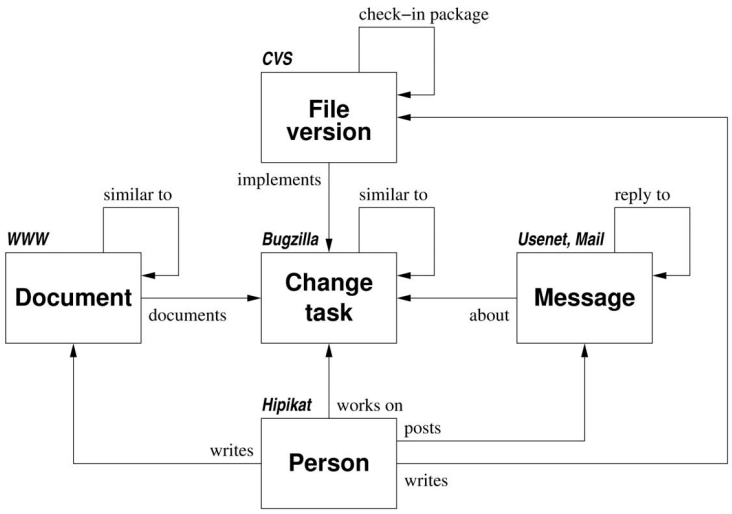


Fig. 1. Artifact types in the Hipikat project memory and the relationships between them.

available from the project management tools or that is inferred by Hipikat. For example, the creator of a file version checked into the repository is always known from the configuration management tool, as is the author of a newsgroup posting. Hipikat infers links by combining information contained within the project artifacts and the metainformation about the artifacts from different information sources. For instance, some links between feature requests and file revisions can be inferred when there is a project convention to include within the check-in comment associated with a 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 versions can be inferred based on metainformation, such as when particular project artifacts were created or modified; for example, it is likely that the author of a bug fix has checked in source code revisions close to the time that the problem report was closed in the issue-tracking system. The specifics of our link inference algorithms are discussed next in Section 4.

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. Those entries also serve as a focus for artifacts in other repositories. For example, source code versions are checked into the source repository fixing a particular set of issues, and newsgroup postings and mailing list messages often contain discussion that either results in a new entry in the issue-tracking system or that is about an existing issue. Other documentation may contain information about a particular entry in the issue-tracking system, such as specific design trade-offs related to a feature request, milestone plans, or regression tests.

### 3.2 Making Recommendations

When selecting and presenting recommendations to a developer, the relationship links are used to determine relevant artifacts in response to a query. The query can be initiated explicitly by the user, or implicitly based on the user's navigation and other actions in the workspace. The two options are not mutually exclusive, although our prototype client currently implements only the former.

The query identifies the artifact that is the "subject" of the query, and optionally can contain additional context or recommendation filtering choices selected by the user or Hipikat as appropriate to the situation. The server receives the artifact key as part of the query, finds the artifact in the project memory, and uses the relationships to find the artifacts to include in the recommendation lists.

For example, when a developer starts working on a feature modification task, the developer may be interested in other change tasks that have a similar description. These artifacts are selected for recommendation by following `similar-to` links (see Fig. 1) and are returned to the user to inspect. (See Table 1 for a full list of artifact types and places in the IDE where a Hipikat query can be made.)

Once the developer has identified a change task that appears to be similar, a query on it leads to source revisions that implemented the task of interest (via the `implements` link). These revisions may help a developer identify code that might 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 might impact the task at hand.

## 4 The Hipikat Tool

We have built a working Hipikat prototype that implements the project memory model and the functionality described above. The prototype has been designed to adapt easily to any project that follows the common open-source development model [26] and that produces at least a subset of the artifact types contained in the project memory schema.

The Hipikat prototype is a client-server system. The client and the server communicate over a SOAP RPC protocol [27], with the recommendations returned by the server in an XML format, described next in Section 4.1. The characteristics of the protocol allow language and platform-independent implementation. This allows a client to be written that is appropriate to a particular project and the development tools used by its members.

TABLE 1
Places Where a Hipikat Query Can Be Performed in Eclipse and the Artifact Type on Which the Query Is Invoked

| Location in the IDE | Artifact type |
|---|---|
| Bug report open in the Bugzilla editor | change task |
| CVS-managed file open in the Java editor | file version |
| File in the CVS Repository view | file version |
| Revision in the CVS Resource History view | file version |
| CVS-managed file in the workspace Navigator | file version |
| Item recommended in the Hipikat Results view | *item's type* |
| Bugzilla search match in the Search results view | change task |
| Java class or method in Outline and Hierarchy views | file version |

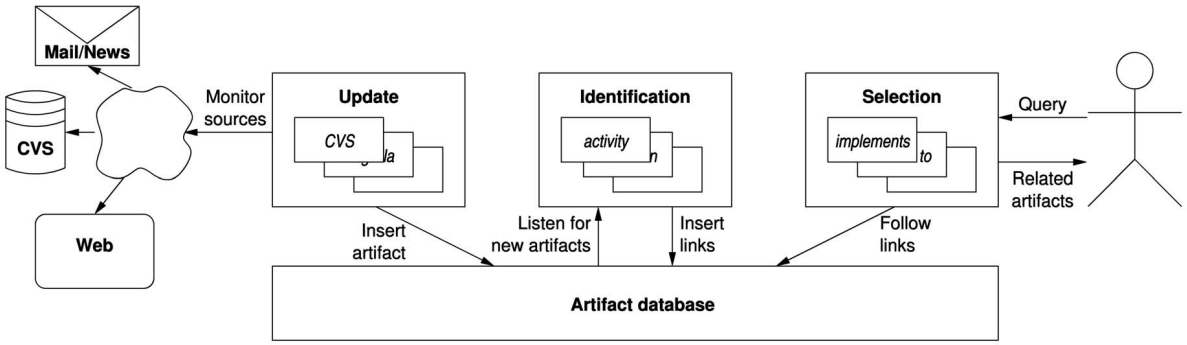


Fig. 2. Hipikat server architecture.

### 4.1 Hipikat Client-Server Protocol

The client issues to the server a request for recommendations, and displays returned results to the user. The client request identifies the artifact for which related items are sought and identifies the user anonymously.[6] The server replies with a list of matches that the client then formats and presents in human-readable format. Each item recommended by the server is represented by a tuple of four values: a key that uniquely identifies the artifact and is used if the user decides to open it or make a subsequent query on it; a human-readable description of the artifact (e.g., a file version's check-in comment); the reason for recommending the item; and the server "confidence," or relative strength of this relationship. The confidence value can be descriptive, as in "High—checked in within five minutes" for a file version's link to a bug report, or numeric in the case of a text similarity measure. The reason for a numeric value for text similarity is because values are not always comparable across sets of recommendations and, therefore, difficult to discretize. Instead, we chose to simply use the value calculated by the similarity algorithm with the expectation that Hipikat users would, with time, develop a sense for the relevance of bugs based on a combination of the value of their own similarity, their rank in the list, and the similarity value of other bugs in the list.

### 4.2 Hipikat Server

The server implements three distinct functions:

**Artifact store update**. The project's archives must be monitored for additions and changes that result from the development and evolution of the system, and the project memory must be updated accordingly to reflect the additions and changes.

**Link identification**. As artifacts are added to a project's memory, the links between related artifacts must be identified and added to the memory. These additions might cause changes or deletions of the existing links for some relationship types (e.g., text similarity).

**Recommendation selection**. In response to client queries, relevant artifacts must be selected for recommendation and returned to the caller.

As Fig. 2 shows, each function is encapsulated in a module. Each module is divided into submodules that handle a single artifact type or link inference. The modules do not communicate directly with each other, but instead share access to the database where the artifacts and artifact links are stored.

The server is written in Java. The project memory is stored in a MySQL relational database.[7]

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

7. http://www.mysql.org.

#### 4.2.1 Artifact Database

The artifact database saves primarily the *metadata* from the new and changed artifacts that are needed to establish relationships between the artifacts. Text from the artifact's contents and metadata may be indexed, depending on the artifact's type. The indexed text is used for searching and making similarity comparisons, which we will describe below in the Identification section.

#### 4.2.2 Update

The update module has a separate submodule to handle each different type of project information source, such as Bugzilla, CVS, or the mailing list archive. Each update submodule monitors its information source for changes, as appropriate for its type. For example, a Web site is scanned in the usual Web-spider fashion [28] by starting from the set of known pages (initially just the project home page) and following all links to pages local to the site. New and changed artifacts are inserted into Hipikat's artifact database, and change listeners in the identification module are notified of the updates.

#### 4.2.3 Identification

The identification system determines links between related artifacts and stores them in the database. Links are determined by applying one or more heuristics to artifacts newly added to the database or modified in some way. The identification system in Hipikat is designed to support multiple heuristics. The identification supervisor manages the registration of each heuristic module and their interfaces with the update system.

Each identification module is registered with the update system as a listener for changes on artifact types for which it is responsible. There are currently five such modules (described in detail below): log-matcher, activity-matcher, text similarity matcher, CVS check-in package matcher, and newsgroup thread matcher. When informed of a new instance of an artifact, or a change to an existing artifact, the identification modules attempt to infer links within the implicit project memory, following the schema from Fig. 1. The identification modules are also notified when the update system's periodic update of an information source is finished, in case they need to do any identification postprocessing, such as recalculating the text similarities.

**Log matcher**. The log matcher exploits the convention used by open-source developers that comments entered during the check-in of source code versions (the "log") contain the id(s) of the bug report(s) that is (are) being fixed by the version's changes. The log matcher uses a small set of regular expressions to search for certain phrases and constructions commonly used by project developers (such as, "Fix for bug 1234"). When a Bugzilla id is detected in a check-in comment, the matcher inserts an `implements` link into the project memory to connect the change task with the file version(s) that were checked in. If no regular expressions match the check-in comment, the log matcher does nothing.

**Activity matcher**. The activity matcher tries to complement the log-matcher by taking advantage of natural work patterns used by the developers, rather than loose conventions, such as the check-in comment, that are not regularly enforced. Shortly after the relevant source changes have been checked in, a developer will usually change the status of the corresponding item in Bugzilla (e.g., to mark it "fixed"), or post a comment notifying others of his or her progress. The activity matcher monitors updates of the Bugzilla database and looks for check-ins that are close to (within six hours of) changes of status of an existing Bugzilla item to "resolved fixed." Check-ins are then grouped into likely work units by looking for all check-ins by a given developer within a small time window (six minutes). (A similar strategy was employed by Mockus et al. in their study of Mozilla development process [26], and German [29] in his softChange tool. The time windows used in these tools were three and approximately 10 minutes, respectively. Neither of the two report on the accuracy of their algorithms, but they do claim to have chosen values that seem to work well. We followed this approach in our implementation of the activity matcher.) Versions in each work unit are linked to the change task in which the activity occurred with an `implements` link, which also records the time difference between the check-in and the activity for later ranking and presentation to the user.

**Text similarity matcher**. 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 $i$th term's *global weight*, $G(i)$, indicating its overall importance in the entire collection, and its *local weight*, $L(i,j)$, which depends only on the frequency of terms within the $j$th document. We use a log-entropy [30] 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 in the collection containing term $i$. The document vector is then projected into a semantic space using Latent Semantic Indexing (LSI) [31] to produce a lower-dimensional vector $D_j$ for the $j$th document. In the second phase, the text similarity matcher uses a standard information retrieval vector-space cosine similarity measure [32] to infer `is-similar-to` links between artifacts. Specifically, the similarity between two documents is calculated as $sim(D_m, D_n) = \frac{D_m \cdot D_n}{\|D_m\| \|D_n\|}$. A selection submodule is responsible for using the computed measures to recommend a small set of nearest neighbors to an artifact (currently capped at 15).

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

**CVS check-in package matcher**. This identification module links file versions that were checked in together as part of the same check-in into the repository. It looks for

other versions with the same author and comments that were checked-in within a six-minute window, similar to the grouping performed by the activity matcher. **Thread matcher**. The simplest identification submodule is the newsgroup thread matcher, which looks for "References" headers in newsgroup articles (mandated by the RFC 1036 standard) and reconstructs conversation threads of a newsgroup posting and subsequent replies. When a user chooses a recommended article for viewing, the Hipikat client opens the article in a newsreader using the "news:" URL, which does not allow navigation to preceding and subsequent articles (as opposed to a user interactively opening it from a subscribed newsgroup). Instead, the user can receive the conversation thread in a Hipikat recommendation and open the individual articles from the recommendation list.

An equivalent approach is applied to matching conversation threads in email archives. A standard email message (as defined in the RFC 822 standard) contains a header referencing the id of the message to which it is a reply. Alternatively, email archives with a Web front end often have navigation links to a message's replies. A Web crawler can then be tailored to the front end to take advantage of those links and present them in recommendations.

#### 4.2.4 Selection

The selection phase takes a set of candidate recommendations, orders it, and possibly removes items from the set to generate a refined recommendation list. Selection works by following links from the artifact specified in a client's request to find a set of related artifacts. Similar to identification, selection is designed to support multiple link types. Selection modules 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. In general, each selection module pairs up with one or more identification modules and works with their link types.

Each module provides a *reason* for recommending the artifact and a *confidence* describing the strength of the relationship. If an artifact is reached by multiple links, the selection module will take that into account when giving a reason for the recommendation and a confidence. For example, a bug report can be related to a file revision both by the id match in the CVS log and by the bug activity match. If that is the case, the selection module will use only the CVS log id match as the reason for the selection, since it has higher confidence than the bug activity match. Higher confidence is assigned to the log id match because it is explicitly entered by the developer doing the check-in, so when it is present, it almost always indicates the correct relationship between revision(s) and bug report(s). (For a complete description of how reasons and confidences are calculated, see [33].)

### 4.3 Hipikat Client

We have implemented a Hipikat client as a plug-in for the Eclipse IDE. One of Eclipse's primary design goals was ease of extensibility, which means that the Hipikat client appears seamlessly integrated into the IDE, and can be used with other software engineering tools plugged into Eclipse. For example, an Eclipse developer can access from the same Search dialog both the Hipikat search and the Java search feature that is bundled with the default Eclipse distribution, simply by clicking on the appropriate tab in the search dialog.

Fig. 3 shows a screenshot of Hipikat running within the Eclipse development environment. User interaction with Hipikat is kept as simple and unobtrusive as possible: A user makes a query by selecting an artifact in the Eclipse project workspace and choosing "Query Hipikat" from the context menu. The selection can be made from most views in the IDE, such as a list of files in the workspace, a Java editor, or a bug report viewer (Fig. 3a). (See Table 1 for a full list of artifact types and places in the IDE where a Hipikat query can be made.) The server responds with a list of related artifacts, which can in turn be opened and/or used for further querying. The developer can leave the querying cycle to explore the source code or documentation, as prompted by Hipikat's suggestions, and return to issue more queries at any later point.

Additionally, the Hipikat artifact database can be searched based on search terms specified by the developer. As already mentioned, this functionality is accessed through a "Hipikat search" pane in the regular Eclipse search dialog.

The identifier of the selected artifact is passed as the second argument in the request to the Hipikat server, described earlier in Section 4.2. The results of a query or search are displayed in a Hipikat *Results view* within Eclipse (see Fig. 3b). The view lists for each recommendation its type (Web page, news article, CVS revision, or a Bugzilla item), its name, why it was recommended, and—if applicable—an estimate of the closeness of the match. The recommendations are grouped by artifact type and by selection criteria as determined by the identification submodule that reported a link.

Double-clicking on a recommendation in the results view opens the artifact for viewing (Fig. 3a). Bug reports and CVS artifacts are opened within Eclipse; news articles and Web pages are opened in a Web browser. Right-clicking on a recommendation pops up a context menu. From this Menu, the developer can also open the recommended item for viewing; more importantly, it can be used to issue another Hipikat query. Recommendations that the developer considers irrelevant to the current task can be removed from the recommendation list to clean up the results view ("given the thumbs-down"); others can be marked as particularly relevant to the query ("given the thumbs-up"), which moves them to the top of the list. (We plan to use this information in the future to add collaborative filtering capabilities.) Lastly, for a CVS file version, its differences from the preceding revision can be shown in the standard Eclipse "Compare CVS revisions" view. The intent of this feature is to make it easier for the developer to see changes to the code that were made in the recommended revision.

## 5 Hipikat Recommendation Quality Study

To investigate the usefulness and accuracy of Hipikat for providing information relevant to a developer working on

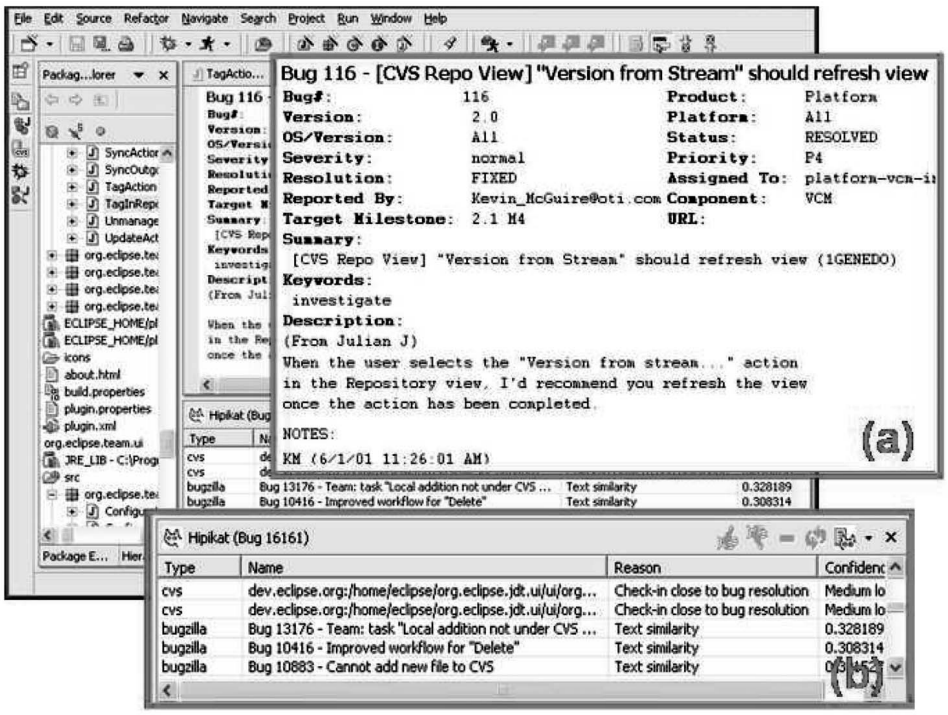


Fig. 3. A screenshot of Hipikat being used within Eclipse IDE during the Eclipse newcomer study (Section 6). Zoomed-in rectangles show a change task, (a) problem report 116 and (b) list of related artifacts.

a software modification task, we evaluated Hipikat's recommendations on a sample drawn from the issue-tracking database of a large open-source software project. We used Eclipse, an extensible integrated development environment that is written in Java and contains around a million lines of code.

### 5.1 Selecting the Sample

Our study targeted modification tasks that were completed for Release 2.1 of Eclipse. Eclipse uses Bugzilla as its issue-tracking database.[8] In general, in open-source software development an issue being tracked in Bugzilla that is marked *fixed* corresponds to a single modification task that was completed by a developer. We thus initially defined the set of eligible tasks as all issues from Eclipse's Bugzilla database that were marked "fixed" between 27 June 2002 (the day Eclipse 2.0 was released) and 27 March 2003 (the day of the 2.1 release). This is the same time period from which we selected change tasks that we used in our other study (Section 6) and, consequently, is the phase of Eclipse development with which we are the most familiar and for which we can best evaluate the relevance of Hipikat's recommendations.

We further narrowed the set of eligible modification tasks to those that a newcomer may have been assigned. Although there are no clear rules that can be used to identify automatically such modifications, a good approximation is to use the *severity* field of the Bugzilla issue. A severity value of *minor* is defined in Eclipse.org's online help for Bugzilla[9] as a "minor loss of function, or other problem where an easy workaround is present." From anecdotal evidence gathered through observing activity in Eclipse's bug database, we noticed that newcomers on the Eclipse team were frequently assigned bugs of this severity during their first few weeks. We therefore concluded that if a bug's severity has been set to the value of "minor," it is usually a good indication that fixing the bug is an appropriate task for a project newcomer and, for the purposes of this investigation, it is the most practical method to select such tasks.

A total of 215 bugs from the Eclipse Bugzilla database matched our criteria (bugs of severity "minor" that were resolved to "fixed" between 27 June 2002 and 27 March 2003). From this set, we randomly selected a sample of 20 bugs for investigation. As we investigated these bugs, we found that some did not represent modification tasks. In these cases, we discarded the bug and drew another one. (One reason nonmodification bugs were present was that the bug was inappropriately marked as "fixed"—for example, if in reality the problem described in the bug was determined not to be a real problem in Eclipse. There is a separate bug status to indicate this situation, called "invalid," but occasionally developers do not use it when they should and mark the bug "fixed" instead.) We also discarded a selected bug if we could not determine how it was fixed, which usually was because it had been fixed as a side-effect of another (larger) modification task. The solution that was accepted by the Eclipse team for the Release 2.1 formed the base against which we compared Hipikat's recommendations.

8. In the Bugzilla terminology, all issues tracked by Bugzilla are called "bugs" even though they can comprise new features, re-engineering, and other development activity. We will follow this usage of the term bug in its all-inclusive sense in this article.

9. https://bugs.eclipse.org/bugs/queryhelp.cgi#severity.

### 5.2 Evaluation Criteria

Since we knew which source files were relevant to each of the bugs in the sample, based on the fix that was eventually implemented, we could evaluate how useful Hipikat's recommendations would have been by looking at the overlap between the files it recommended and the actual solution set. This is a simplified measure of recommendation quality because it does not take into account information contained in other artifact types that Hipikat recommends, such as bugs and newsgroup articles, but it gave us at least a conservative indication of Hipikat's usefulness for finding the relevant information.

The performance of information retrieval systems is often evaluated in terms of recall and precision [32]. Informally, *precision* is the proportion of retrieved material that is actually relevant to the query, and *recall* is the proportion of relevant material that is actually retrieved.

We consider as the relevant material in this case the files checked into the CVS repository that can be reached through the two-step investigation procedure of querying Hipikat first on the assigned task, and then on bugs in the returned recommendation list that were marked as "fixed." This was the same search strategy that was used by the newcomer participants in our other study (see Section 6.3.3), and we therefore believe that this evaluation of the relevance of files recommended by Hipikat is a good approximation of the helpfulness of Hipikat to a newcomer.

More formally, we define the *solution* of a modification task $m$ to be the set of files that Hipikat returns as the implementation of the task; we denote this as $f_{sol}(m)$. The Hipikat recommendation $\mathcal{H}(m)$ is the set of all project artifacts that Hipikat returns in response to a query on bug report $m$. We then define the set of recommended files $f_r(m)$ as the union of solutions to completed modification tasks $b$ in $\mathcal{H}(m)$:

$$f_r(m) = \bigcup_{b \in \mathcal{H}(m)} \{f_{sol}(b) \mid b \text{ is a completed modification task}\}.$$

The precision of a set of recommended files $f_r(m)$ is thus the fraction of recommendations that did contribute to the files in the solution $f_{sol}(m)$ of the modification task $m$:

$$precision(f_r(m)) = \frac{|correct(f_r(m))|}{|f_r(m)|},$$

where

$$correct(f_r(m)) = f_r(m) \cap f_{sol}(m).$$

The recall of a recommendation is the fraction of the files in the solution $f_{sol}(m)$ of the modification task $m$ that are recommended:

$$recall(f_r(m)) = \frac{|correct(f_r(m))|}{|f_{sol}(m)|}.$$

However, there is more value in Hipikat recommendations than just finding the location where the solution should be implemented. The usefulness of recommendations may lie in their *examples of use* of the relevant APIs, rather than pointing to the class that contained the solution. We therefore extend the definition of the solution to a modification task with the set of *constructs* $c_{sol}(m)$ that were part of the implementation. These constructs can include method calls or specific API use patterns in the case of Java classes, or portions of the XML files that are used to define Eclipse plug-ins and their connections, for example. However, we will not calculate the precision and recall for constructs because precisely defining the granularity and number of relevant constructs in a given solution is a matter of individual judgement. Instead, we will describe the relevant constructs and their presence in Hipikat recommendations in the detailed description of results for the three representative bugs in the next section.

### 5.3 Results

The results of this investigation are summarized in Table 2. For each bug report in our sample, we list the precision and recall of recommended files. In each case, we also include the rank of the first recommendation that contained the right files or construct, respectively. The "Average" row gives the mean and median for precision and recall of the sample. The "First-useful" column indicates the rank of the first recommendation that pointed to the right files or construct. For both files and constructs, we give two numbers: the "All bugs" column indicates the absolute rank among all recommended bug reports, while the "Completed" column gives the rank when taking into account only those recommendations that represented completed modification tasks (i.e., the ones marked "fixed") that also had attached file revisions.

We will return to Table 2 in the next section (5.4), when we discuss our study results in more detail. But first, in the remainder of this section we describe in detail recommendations for three specific bugs from the sample. These bugs were selected as examples of a situation in which Hipikat made very good, moderately useful, and poor recommendations, respectively. We use them to illustrate the kind of help that can be expected from Hipikat in practical situations.

#### 5.3.1 An Example of a Very Useful Recommendation

Bug report 23,719 points out an inconsistency in names of automatically generated getter and setter methods when the underlying field is a boolean with a name like "isFoo." In that case, the getter name will be "isFoo," but the setter will be automatically named "setIsFoo" instead of the correct "setFoo," which causes interoperability problems with development tools that depend on this common naming convention.

**Solution**. The solution of this bug involves several classes. The core is in the `proposeSetterName` method of class `NameProposer`, which, if the field is a Boolean whose name starts with "is," removes the "is" before adding the "set" prefix to the field name. The changes to this method mean that an extra Boolean parameter has to be added in its callers. Last, a JUnit test was written and added to the suite of tests for the JDT UI plug-in.

**Recommendations**. Hipikat can help solve this bug because it provides a highly ranked recommendation that points out the relevant location and constructs. The top Hipikat recommendation for bug 23,719 is bug 6,887, which corresponds to the modification task that changed the name

TABLE 2
Recall and Precision of Recommendations for a Sample of 20 Bug Reports

| Bug id | Files | | First-useful recommendation | | | |
|---|---|---|---|---|---|---|
| | Precision | Recall | Files | | Constructs | |
| | | | *All bugs* | *Completed* | *All bugs* | *Completed* |
| 23719[a] | 0.56 (5/9) | 0.71 (5/7) | 1 | 1 | 1 | 1 |
| 28382 | 0.50 (1/2) | 1.00 (1/1) | 5 | 1 | None | |
| 26338 | 0.17 (1/6) | 1.00 (1/1) | 2 | 1 | 2 | 1 |
| 30943 | 0.15 (4/26) | 1.00 (4/4) | 4 | 3 | 4 | 3 |
| 23321 | 0.13 (2/16) | 1.00 (2/2) | 3 | 1 | 3 | 1 |
| 27822 | 0.11 (1/9) | 1.00 (1/1) | 12 | 2 | None | |
| 19857 | 0.09 (2/22) | 1.00 (2/2) | 3 | 2 | 2 | 2 |
| 3248 | 0.07 (1/14) | 1.00 (1/1) | 10 | 4 | None | |
| 2338 | 0.06 (1/17) | 1.00 (1/1) | 7 | 3 | 10 | 4 |
| 9615 | 0.06 (1/17) | 0.50 (1/2) | 2 | 2 | 2 | 2 |
| 34641 | 0.05 (1/19) | 0.50 (1/2) | 10 | 4 | 10 | 4 |
| 24168 | 0.04 (1/24) | 1.00 (1/1) | 2 | 2 | None | |
| 20017 | 0.04 (1/24) | 0.25 (1/4) | 8 | 2 | 8 | 2 |
| 23365 | 0.04 (1/25) | 1.00 (1/1) | 1 | 1 | 1 | 1 |
| 22260 | 0.04 (1/26) | 1.00 (1/1) | 4 | 1 | None | |
| 28513 | 0.04 (1/27) | 1.00 (1/1) | 4 | 3 | 2 | 2 |
| 6732[b] | 0 (0/26) | 0 (0/1) | N/A | N/A | 5 | 1 |
| 31972 | 0 (0/36) | 0 (0/1) | N/A | N/A | None | |
| 32067 | 0 (0/14) | 0 (0/1) | N/A | N/A | 2 | 2 |
| 33182[c] | 0 (0/60) | 0 (0/2) | N/A | N/A | None | |
| **Average** | 0.11; 0.06 | 0.65; 1.00 | | | | |

of automatically generated getter names for Boolean fields from "get*field*" to "is*field*." Again, in this case, the core of the task's solution was in the `NameProposer` class, but this time in the `proposeGetterName` method.

**Applying the recommendations**. The top Hipikat recommendation for bug 23,719, thus, contains all constructs needed to solve the new report. These constructs will have to be applied in a different method of class `NameProposer` (`proposeSetterName` as opposed to `proposeGetterName`), but that difference should be fairly obvious from the two operations. The recommendation also points out all of the other source code files that will have to be changed as part of the implementation, although this would be evident from compilation errors caused in those files by the changes in `NameProposer`.

The only aspect of the real fix to bug 23,719 that the recommendation missed is the introduction of the corresponding JUnit tests. However, the code recommended by Hipikat in this case should be sufficient for even a newcomer to implement the functionality. Given the complexity of the API for manipulating the model of Java programs in Eclipse, this is very helpful.

#### 5.3.2 Moderately Useful Recommendations

Bug 6,732 points out that expanding a node in the tree widget in the *Navigator* view can take a long time when a lot of elements are present. The reporter of the bug suggests that the widget follow accepted HCI practice by showing a busy cursor while node expansion is in progress.

**Solution**. The solution is to perform the node expansion operation using the method `showWhile(Runnable)` of class `BusyIndicator`, with the operation encapsulated in a `Runnable` interface.

**Recommendations**. Hipikat recommendations for this bug do not point to the location of the solution. However, the first bug in the recommendation list that is marked "fixed," bug 2,937, contains the exact constructs to implement the busy cursor, although in a different context. In total, three out of four bugs in the recommendation list that were marked "fixed" lead to the relevant constructs

(bugs 2,937, 15,506, and 9,687). The remaining completed modification task that was recommended but not useful (bug 3,790) is linked, with low confidence, to three file revisions that seem to be irrelevant to the task. (It appears that in this case the developer in charge did not use any of the practices that allow Hipikat to make the connection between CVS check-in comments and Bugzilla items.)

**Applying the recommendations**. Arguably, after reading these recommendations, a newcomer should have no trouble realizing *how* to implement this modification, although finding the exact location (AbstractTreeViewer) might be challenging.

#### 5.3.3 Unhelpful Recommendations

Bug report 33,182 is an example in which Hipikat did not provide any help, either to locate the file(s) involved, or to identify the constructs necessary for the solution. The topic of the report is the inclusion of certain sections into the so-called "update preview" display even when they contained no information. The solution that was implemented is very simple and is contained in a couple of setter methods in a Java class, with the text messages externalized in a properties file for easy localization of the user interface.

There are two related reasons for the lack of useful recommendations. One is that the problem report itself is very terse. (This seems to be the case for most problem reports that we have come across in this particular subsystem, "Platform-Update".) Its contents are fairly generic to the subsystem and, so, none of the recommended problem reports are truly similar, although 10 of them are from the area. In addition, programmers working on this part of the code rarely enter CVS check-in comments, which makes recommending much more difficult for Hipikat. (As an illustration, out of 71 revisions of the class containing the solution to this bug, 63 have no check-in comment at all, four had only the number of the bug being fixed by the revision, and one had a two-word description of the feature implemented in the revision.) Given that, at the time, almost all work in this area was done by only two developers and that there were no other modules that depended on it, it is perhaps not surprising that Hipikat did not perform well in this kind of environment.

### 5.4 Discussion

Table 2 shows that in 12 modification tasks, out of a sample of 20, Hipikat was able to point out all of the files involved in the task's solution (recall 1.0). In three more cases, Hipikat recommended half or more of the relevant files. We did not numerically evaluate Hipikat's performance on recommending constructs useful for the solution, but, in our sample, there are two cases when Hipikat successfully identified constructs even when it did not identify files (bugs 6,732 and 32,067). Intuitively, this will be true even with a perfect recommender if new functionality is being introduced into a module that is already present in other parts of the system (as in our example of a moderately successful Hipikat recommendation, bug 6,732).

Just as importantly, useful recommendations are usually ranked high among completed modification tasks. Relevant files and constructs are found from the first or second top-ranked completed modification in 11 out of 16 cases for files and 10 out of 13 times for constructs—and always within the top four. This is important because we observed in the Eclipse newcomer study that the participants were reluctant to investigate too far down the list.

Rankings of useful recommendations are lower when all bug reports included in Hipikat recommendations are taken into account. However, users know that bug reports that have not been marked "fixed" do not need to be further investigated to look for associated code. In this respect, we expect the developer to make sensible choices when investigating Hipikat recommendations, which we believe is a reasonable assumption. Hipikat recommends all bug reports, not just the ones marked "fixed," in case they contain interesting discussion, for example, mention of other bug reports or reasons why the report will not be considered for implementation.

Although Hipikat's precision in this study was quite low, it is part of the inevitable trade-off to get better recall. Again, we feel that it was the right choice to make; there is a lot of useful information that developers get just from seeing the name and module of a file, and they can use this information to quickly filter the recommendations for those most likely to be really relevant to the task. We saw this behavior in the participants in the Eclipse newcomer study described in the next section, and it has also been observed in other studies of programmer strategies of code reuse [13]. However, giving a smaller number of recommendations and using different criteria to select an artifact for recommendation remain open areas of research.

## 6 The Eclipse Newcomer Study

In addition to investigating Hipikat's performance on information retrieval measures, we also observed how it was used by programmers under realistic conditions in an earlier study that we summarize here. We focused on the following three questions:

1. Can newcomer software developers use information from the project memory about past modifications completed on the project to help them in a current modification task?
2. When and from which artifacts will newcomer developers who are working on a software change task query Hipikat?
3. How will the newcomers evaluate Hipikat's recommendations and how can they utilize those recommendations in their tasks?

To ensure that we studied realistic participants working on realistic tasks, we selected a large open-source software project for which a full history of changes to the code, developer discussions, and problem reports were publicly available. As in the recommendation quality study (Section 5), this project was the Eclipse IDE. We then selected two previously completed enhancement requests from the Eclipse issue tracking database as tasks in the study. In selecting the change tasks, we were looking for modifications that were complex enough to challenge the study participants and to require serious effort to understand the problem and come up with a solution. By choosing enhancements to an earlier version of Eclipse,

we were able to devise a set of correctness criteria based on the solutions adopted by the Eclipse team. We could then check the participants' solutions against the correctness criteria. We created a copy of the Eclipse project artifacts as they existed at the time when the enhancement requests were made, and formed an instance of the project memory on this copy.

Because we were interested in studying *newcomers*, not novices, the study participants were required to have adequate programming experience in the programming language of the system under study (that is, Java), but they had to have no previous experience as developers on the system itself. We also required participants to have had experience developing large or medium-sized software systems, and to be familiar with issues involved in working on such systems, as well as tools commonly used to manage projects of such size (e.g., configuration management or issue tracking systems). This made the pool of potential participants much smaller, as we could not use easily recruitable computer science undergraduates who would have had insufficient knowledge and experience.

Given the questions asked in this research and the complexity of the tasks analyzed, we determined that a case study was the appropriate methodology for this stage of the research, using multiple cases to try to capture individual working styles. Because we wanted to look in detail at how developers accessed Hipikat and used its recommendations while working on modifications to a new software system, we ruled out a controlled experiment. Instead, we chose a largely qualitative analysis that would allow us to look for patterns across the cases and handle large individual differences among the participants in programming and exploration styles.

Because we wanted to be able to compare the solutions with each other, all participants worked on the same set of tasks. We also wanted to compare the newcomers' end products with those of experienced developers who worked on the project, so we recruited several members from the Eclipse development team and asked them to work on the same tasks and serve as our baseline for comparison. This design allowed us to study Hipikat under conditions similar to those faced by newcomers to many large open-source systems, to test the system on real tasks, and to compare the results of the newcomers using Hipikat with those of experienced team members not using Hipikat and with the "correct" solution of the bug.

## 6.1 Design

Each participant worked on two change tasks, which we describe below in more detail. One task was easier than the other. The order in which the participants worked on the tasks was randomized to control for learning effects.

### 6.1.1 Easy Task

This modification request[10] described a need, when the mouse is hovering over a particular point in an editor for Java source code, to display a breakpoint's properties in a pop-up window. The request initially asked for displaying a few basic properties, such as the breakpoint's line number. A subsequent comment in the request's discussion suggested displaying an additional property of a breakpoint: whether it stops the execution of the entire VM or just the current thread. The participants were told that the latter was an optional property that they could implement if they so chose.

### 6.1.2 Difficult Task

The second modification request[11] involved the interaction of a developer with the UI during versioning operations on a group of files. A file can be in one of three states: new (no versioning information), versioned in the repository, or ignored for all versioning operations (typically temporary and automatically generated files). The request noted that committing a new file to the repository requires two steps: "adding" to mark it as a versioned file and then "committing." It asked for a more intelligent handling of this case, similar to the way it is done through the "synchronization view," where new files can be automatically versioned when they are committed, if the user chooses to do so.

## 6.2 Procedures

Because of the time required of each participant, the study was divided into two sessions, training and programming, that took place within three days of each other, depending on the participant's schedule. The experts did not need the training session and had their programming session on a weekend, to avoid interference with their regular jobs.

Each of the eight newcomers underwent four hours of "hands-on" Eclipse training. The participants individually worked through three online tutorials that included frequent exercises applying the covered material. The tutorials covered using Eclipse, programming Eclipse plug-ins, and using Hipikat. The participants worked on their own, but the researcher was present in the room to answer any questions. The four experts did not go through any training because they all had significant experience with Eclipse and did not have Hipikat available during the programming session.

The programming session was divided into two parts, one for each change task. The maximum time allowed for each task was fixed at two and a half hours. The participants were asked to complete a change plan and describe it to the experimenter before proceeding with implementing the change. Once the change plan was completed, we conducted a semistructured interview in which we asked both about the details of the plan and the process used to come up with it, including tools used and information accessed. At the end of the task, we conducted another semistructured interview where the participant showed us the details and described the process of implementation. During this interview we asked critical incident-type open-ended questions about the most difficult part of solving the task and how the participant went about solving it.

We collected all code modifications that the participants made while they worked on each task. These were checked for correctness against a set of criteria that we had identified previously. These criteria—shown in Tables 3 and 4—are

10. Request 6,660 in the Eclipse problem report database.

11. Request 20,982 in the Eclipse problem report database.

TABLE 3
Correctness Criteria for the Easy Task

**Hover:** Displaying new properties in the hover.

- Identify JavaLineBreakpoint and JavaBreakpoint classes
- Identify markers and the message attribute
- Setting the marker message

**Updating the hover:** Changing the breakpoint's properties should be reflected in the hover.

- *Hover updated when user toggles suspend policy*
- *Hover updated when user changes the hit count*
- *Hover updated when user changes the condition*

**Subclasses.** New hover functionality should work for subclasses of JavaLineBreakpoint.

- *Method breakpoint hover is correct*
- *Watchpoint hover is correct*

**Style:** Modifications should respect existing architecture and coding style.

- Suspend policy is added in JavaBreakpoint, line number in JavaLineBreakpoint
- Strings are externalized

*Italicized criteria are the special cases referred to in Section 6.2.*

sufficiently abstract to cover the required functionality of added features, but still allow variation within the actual implementations. We also included special cases that are not always covered explicitly in the feature request description, but would result in bugs under certain circumstances if they were not recognized (shown italicized in the two tables). Last, we required that the added code be readable and maintainable, and that it follow the Eclipse team's coding practices.

## 6.3 Results

### 6.3.1 Easy Task Solutions

All of the participants implemented the basic requirements of this task: displaying a pop-up window with the breakpoint properties on mouse hover. However, many participants did not handle the special cases properly, which introduced bugs into their solutions. Surprisingly, this was even more the case among the experts, where only one of the four participants (25 percent) correctly updated the information in the pop-up after the breakpoint's properties were changed by the user. Of the eight newcomers, three participants (38 percent) handled this correctly, and two more (for a total of 63 percent) handled it correctly within the scope of basic range of properties that they chose to implement (that is, the line number, condition, and hit count).

The cause of nearly all faulty solutions was missed method inheritance interactions between the concrete and abstract breakpoint classes. An examination of the plans created by the expert participants who failed to handle those special cases shows that all of them talked exclusively about the concrete subclass. Conversely, the newcomers' change plans regularly mention both classes. We believe that the newcomers did so well because both classes were included in the Hipikat recommendation from which they were starting and, so, they were used to thinking about the two classes as a single unit, which was reflected in their plans and implementations. This is a good example of a valuable bit of information that was never explicitly written down anywhere in the project artifacts and, yet, was implicit in the links between the artifacts and became obvious to the newcomers during their exploration of the memory. Without viewing Hipikat's suggestion, it was not at all obvious that both classes would need to be updated. Indeed, half of the expert participants overlooked it, causing bugs in their solutions.

### 6.3.2 Difficult Task Solutions

The participants were less successful in solving this task. Although the two groups have virtually the same group average score (8.4 out of the 15 criteria for the experts and 8.6 for the newcomers), the expert group arguably performed better on the basic requirements of the task: detecting new uncommitted files, displaying the message dialog to the user, marking them as versioned when directed by the user, and proceeding to commit to the repository. Three of the four experts (75 percent) solved these requirements correctly. The unsuccessful expert was completely on the wrong track with her planned solution and did not implement any of these steps. Thus, the expert group's average score is partly skewed lower by the one unsuccessful member; however, that's not the sole reason, since the other three experts' solutions—although solving the basic requirements of the task—still did not handle correctly a number of special cases, and had an average score of 10.7 (71 percent).

TABLE 4
Correctness Criteria for the Difficult Task

**Menu:** Operations in the versioning context menu should be grayed-out as appropriate.

- Identify *CommitAction.isEnabled* method
- Enable the "commit" option even if some of the selected resources are new
- *Do not enable the "commit" option if all resources are in .cvsignore*

**Detecting selected new files:** When the "commit" option is selected in the versioning context menu, the code should check for the existence of any new files in the active selection.

- Identify *CommitAction.execute* method
- Look for new resources in the active selection
- *Check in selected subdirectories for new resources (any number of levels deep)*

**Message dialog:** If there were any new resources in the active selection, a dialog should be put up to ask the user whether to add them to CVS management before proceeding with the commit.

- Put up the dialog if any of the selected resources are new
- If the user presses OK, add the new resources to CVS management
- *If the user presses No, do not do anything to the new resources, but proceed with the commit process*
- *Cancel aborts the whole commit process*

**Committing new resources:** If the user selected "yes" or "no" in response to the message dialog, the rest of the commit process should proceed appropriately.

- Show the check-in comments dialog
- All selected managed resources are committed to the repository (including the new ones)

**Style:** Modifications should respect existing architecture and coding style.

- Do not simply call *ForceCommitSyncAction* to display the dialog (increases coupling)
- Externalize message in the dialog
- Use the *visitor* pattern to look for new resources in the selected subdirectories

*Italicized criteria are the special cases referred to in Section 6.2.*

In the newcomer group, three of the eight participants (38 percent) managed to implement all of the basic requirements (average score 11.7, or 78 percent). Two more newcomers were able to detect the new files and displayed the required message dialog to the user, but then did not implement marking those files as versioned. Of the other three newcomers, one's solution was almost correct but for a runtime error, one still had syntax errors in the code when the time ran out, and the last one did not get beyond correctly identifying the methods where his solution should go. In this respect, all of the newcomers got farther than the unsuccessful expert since even their incomplete solutions were on the right track.

The participants had more difficulty with the special cases in this task. For example, none of their solutions looked within directories that were being committed to check whether they contained any new files. The newcomers should arguably have been aware of this special case. Detecting these files during the commit operation was discussed and accepted as desired in an earlier problem report when the corresponding feature was being added in the "Synchronize view." This problem report was recommended to them by Hipikat—ranked highly in the recommendation list for its similarity to the assigned task—and they even used it as the basis of their solutions. However, it was easy to overlook this point, buried as it was in the middle of a lengthy discussion within the problem report.

### 6.3.3 Accessing Hipikat

Not surprisingly, Hipikat was accessed less during the easy task than during the difficult task. An average of 3.6 and 6.3 queries were made, respectively.[12] Almost all queries

12. Statistics in this section refer to unique queries made in each task. Occasionally, participants queried on the same artifact more than once during the course of a task. Because those repeat queries were probably used as an alternate query "history" mechanism, we did not count them when calculating the averages.

were made within the first hour, especially when a participant was successful in formulating a solution plan. (Slightly less than 20 percent of all queries were made after the first hour, or 0.7 and 1.1 query in the easy and difficult task, respectively.) Once a participant knew the file(s) to be changed and had determined a general plan of how to implement the change, he or she did not make any more Hipikat queries. Hipikat was apparently used as a tool to help get an initial understanding of the assigned task, but not in the execution of the task.

#### 6.3.4 Evaluating and Using Hipikat's Recommendations

Based on the query and access patterns described above, it appears that the participants' exploration was focused on solving the assigned task, rather than gaining deeper understanding of the code. Furthermore, their exploration appears to be defined by a sequence of subgoals. The most important thing was to find code relevant to the current subgoal. The strategy followed by the newcomers was to use Hipikat's recommendations to find code that could be reused in the task's solution and/or the probable location of where the solution should be implemented. However, using the recommendations poses its own challenges. First, a potentially useful recommendation has to be recognized. Then, the recommended piece of code has to be understood in terms of what it does and how it fits into the larger system. Finally, that code has to be adapted to its new purpose, which may involve moving it to a different place in the system's architecture.

We found that the crucial first condition to recognizing a useful recommendation was whether the description of a problem report looked "interesting." That is, it had to be similar enough to the current task to make it likely that the associated code could be reused, or at least that the participant could learn from it information relevant to the task. Not surprisingly, participants searched for similar reports by going down the list of recommendations returned in response to the query on the task's problem report. Given the effort needed to understand the code associated with more complex recommendations, we observed reluctance to investigate too many recommendations down the list. If anything, we noted that participants tended to stop their exploration as soon as they had a starting point from which to look at source code.

In some cases, such as in the top recommendation in the easy task, the code in the recommended revisions was easy to understand just by seeing the modified lines highlighted by Hipikat. At this point, the participant would switch from the Hipikat view to working with the source code directly in order to understand it more fully, and especially how this code interacted with the rest of the system.

In other cases and, in particular, in the difficult task, this could require significant effort. For instance, some highly-rated recommendations in the difficult task included up to nine files that were implementing the fix for a problem. Understanding just how changes in those nine files were related to each other, what exactly they do, and which of them were relevant to the actual task was a serious challenge. The way Hipikat presented the recommendations involving file revisions was not sufficiently helpful in such cases. A common "shortcut" used in such situations by the study participants was to consider the names of the files included in those revisions as an indication of their potential relevance, and to switch to viewing source code even if the revision's changes were not quite understood. Participants preferred to build their understanding of such files from scratch by reading it in an editor, at the risk of following a false lead and having to return to searching.

## 7 Discussion

### 7.1 Model

Our approach depends on the existence of extensive repositories of software development artifacts. This dependence on artifact repositories is not far-fetched in the modern world of software development, both open-source and commercial. In this section, we discuss some of the open issues regarding certain aspects of the Hipikat model, but do not question the basic assumption that those artifact repositories exist in the first place.

#### 7.1.1 Unit of Recommendation

Hipikat makes recommendations at the artifact level of granularity; that is, what is recommended is a bug report, a file revision, a Web page, etc. This level of granularity follows naturally from artifact sources and usually results in recommendations that are logically self-contained. (For example, even though there are multiple individual comments in a bug report, they are all related to a single bug.)

In some cases, it may be desirable to recommend only a portion of an artifact, such as when only a couple of comments in a long bug report are relevant to a query. This could be accommodated in the existing Hipikat model by recommending entire artifacts and highlighting the most relevant passages, similar to the way Google highlights the search terms in cached pages. The highlights could be represented uniformly and independently of the artifact type as a collection of ranges of text in the artifact. This kind of relevant passage detection could be automated for artifacts consisting of natural language text (i.e., not computer code) by using existing information retrieval techniques for topic detection within documents (e.g., [34]). For a discussion of manual identification of such passages, see the later section on collaborative filtering (Section 7.1.3).

At other times, the unit of recommendation should be a set of mutually related artifacts, such as a set of file versions that were checked in together to fix a bug. It makes sense to think of all files in the set as a single change; trying to understand modifications in one class in this set (if the files are Java source code, for example) will be futile without taking into account code modified in other classes in the set. A general way in which such *sets* of artifacts could be recommended as a group would be by returning a hierarchy of recommendations, rather than the flat list format we currently use. That way a single recommendation could either point to a single artifact, or (recursively) contain sets of recommendations.

#### 7.1.2 Better Time Awareness

The value of information contained in artifacts usually diminishes with time as the system changes. Therefore, even if an artifact were the best match to a query on the basis of text similarity, it will not be very useful if it describes the functionality of an old version of the system that no longer works that way. (Even worse, it may be especially misleading to the novice who does not know enough project background to recognize that the information is out of date.) Some existing recommender systems attempt to model this information decay by decreasing the calculated similarity value by a factor that takes into account the age of the document under consideration (e.g., [35]). The challenge inherent in this approach is that the time decay factor is essentially arbitrary, chosen by trial-and-error, and not necessarily portable across different document collections.

However, not all information about a software system gets out of date at the same rate. If a module has not changed much over time, even old artifacts related to it will still be relevant. Information does not decay monotonically as it ages: A system whose architecture has degraded over time can be refactored to restore it and reverse the entropy. Therefore, dealing with this issue remains an open research question.

#### 7.1.3 Collaborative Recommendation

Hipikat's recommendations are currently made purely based on the content of artifacts. However, given a large enough user base, it becomes possible to make use of other user's interactions with the tool when choosing artifacts to recommend. Such *collaborative recommending* uses ratings given to artifacts by other users to select the ones that are the most relevant to a given query. One way to obtain the ratings is *explicitly*, asking the user to manually rate the artifacts. We have implemented this approach in the current prototype as the "thumbs-up" operation in the *Hipikat results view*: Giving a thumbs-up to a recommendation means that in the future it will be ranked nearer the top of recommendations when it matches a query. This is a very simplified form of collaborative recommending, and one that was intended primarily as a proof-of-concept. A more complete implementation would take into account the closeness of the current query and the query (or queries) for which a recommendation was given the thumbs-up or the similarity in user profiles (e.g., interests or expertise) between the current users and the users who created the ratings. Ideally, collaborative criteria should be integrated into a single recommendation model with the content-based criteria (cf. [36]).

Using explicit ratings has a significant drawback, as it requires the users to perform additional work which is not going to directly benefit them, which, as Grudin has noted [6], is therefore unlikely to get done. Perhaps even more problematic is the fact that the action of rating a recommendation fits rather awkwardly into the existing software development process: As we have found in our user studies, recommendations are investigated in the initial stages of a change task, but the natural point to rate their usefulness would be once the task has been completed, when the developer can truly evaluate which ones were the most useful. Trying to rate recommendations early on, as they are investigated, not only introduces room for error, it is actually disruptive to the user's main task—understanding and planning the software change.

Typical collaborative recommenders that use explicit ratings do not suffer from this drawback because they naturally fit the act of rating into an existing process: people *like* giving their opinions on movies (e.g., [37]), music (e.g., [38]), or books (e.g., the reader reviews on Amazon.com). In Hipikat's target domain, we argue that this is not the case; an approach that uses *implicit* rating (i.e., inferred from users' actions) would be more appropriate. While the best way to obtain these ratings for Hipikat needs to be empirically established, two basic approaches that have been used in other recommender domains are applicable to our case. The action of viewing an artifact can be taken as a positive rating (cf. [39]). Moreover, the developers' history of navigation within the IDE and Hipikat can be used to extract patterns in the temporal order of viewing of artifacts (e.g., [40]). By discovering these patterns, the tool can then recommend artifacts that were consistently accessed in similar contexts. The unresolved problem with navigation patterns is how to identify the "dead ends" and distinguish them from useful paths. Especially newcomers exploring the code will likely not have the experience to recognize dead ends; so, by following such irrelevant paths, they will increase those artifacts' future rank, to the detriment of all users.

### 7.2 Implementation

#### 7.2.1 Presentation of Query Results

Both the presentation of Hipikat's recommendation list and its display of recommended artifacts could be improved. Showing matches in a flat list sorted by their relevance is the dominant way of presenting results by recommenders and search engines in general, in a multitude of domains. However, when the user's purpose is exploratory browsing of a collection, such a flat-list presentation does not indicate relationships *within* the results, only to the query itself. If the user can see similarities between individual matches, he or she can identify clusters within the results, making it easier to discard subsets which match the query in ways not relevant to the user's current purpose (e.g., [41]). This task can be made even easier when the clusters are automatically labeled with their most salient keywords (e.g., as in WebRat [42]).

As we have already noted in Section 6.3, even after participants in the Eclipse newcomer study recognized the potential relevance of a recommendation, understanding the recommended artifacts was sometimes more difficult than it could have been. This is particularly the case for file revisions. Even when a developer can see a revision side-by-side with its predecessor and with changes highlighted, understanding how those changes work is still a challenge when the changed code is scattered across many files and

changes need to be correlated to see how they work together and fit into the rest of the system. We believe presenting these changes and relationships graphically would assist the user.

#### 7.2.2 Scaling Up

The main bottleneck in the heuristics that are currently used in Hipikat is using Latent Semantic Indexing (LSI) for determining text similarity. LSI uses singular value decomposition (SVD), which is a costly operation in terms of both memory consumption and computation time. In addition, LSI's precision-recall performance tends to degrade as document collections become very large and heterogeneous [43]. There is, however, ongoing research into scaling up LSI to handle such large selections. For example, Tang et al. report several orders of magnitude better efficiency than LSI, while maintaining LSI's retrieval quality even for large and heterogeneous document collections [44]. Incorporating such a technique into Hipikat should allow it to scale even to projects of the size and duration of Mozilla.

### 7.3 Validation

#### 7.3.1 Types of Artifacts Most Used in the Study

Our studies did not fully explore all artifact types and links present in the project memory. In the study evaluating the quality of Hipikat recommendations (Section 5), we drew our sample of modification tasks from minor severity bugs, which meant that the changes did not require much discussion or elaboration of features. Our measures consequently focused on getting to the relevant files and code constructs.

In the Eclipse newcomer study (Section 6), the two change tasks used in the study, combined with the Eclipse.org project practices, meant that the most useful artifacts were again bug reports and file revisions. In general, bug reports were used to identify similar change tasks done in the past whose solutions could be reused or serve as a springboard for understanding relevant code. The reuse and learning were then done from the associated file revisions, until eventually participants switched to working with the source code directly.

Study participants used other artifact types (Web pages and newsgroups) far less, but it should be emphasized that the issue-tracking system (that is, Bugzilla), as used in Eclipse.org and other open source projects, is not simply a collection of descriptions of how to reproduce a bug or of requests for new features. It serves an important additional purpose as a forum for discussing design rationale and implementation alternatives. As we noted in the study analysis (Section 6.3.4), these parts of bug reports did not seem to be read as carefully or understood fully; the participants appeared focused on finding code useful to get the change task done. We do not believe that this detracts from the study's support of our research claims; however, a closer examination of the usefulness of rationale contained in the project memory and how developers try to access it using Hipikat would be an essential step to be done in future research.

#### 7.3.2 Measure of Effectiveness

The primary measures we used in the Eclipse newcomer study to evaluate the participants' performance were the correctness criteria we identified from the "real" implementations of the two features as they were developed by the Eclipse team and included in subsequent releases of the software. Although we tried to make them as objective as possible, there is always some room for a rater's subjective interpretation.

The correctness criteria should not be treated like a standardized test instrument, or statistical significance expected from the numerical values of participants' scores. The correctness scores were simply a means to monitor the progress of the participants' solutions and help identify situations where using Hipikat was helpful or where it failed. The correctness scores are complemented with more detailed observations of the participants' work, as well as by their comments in interviews. We believe this combination gave us a good picture of the issues we studied. Using an alternative "pure" code quality measure would not have been as helpful in this context; neither would simply focusing on the time needed to solve the tasks. The latter measure is particularly problematic because the only way to meaningfully use it for comparison across participants is if their solutions are all correct. Yet, evaluating partial solutions and pointing out to a participant cases that still need to be handled would provide so much guidance as to completely skew the results even if it did ensure that all solutions were correct.

### 7.4 Impact of Extended Use of Hipikat

Although one of our starting principles was that little or no change to the development process should be required in order to use Hipikat, it would be interesting to see how extended use of Hipikat would affect the developers' behavior. Our study focused on fresh project newcomers, but even if they were the only ones using Hipikat, it is possible that this would affect the development practices of the entire team, once most of its members had used it during their "apprenticeship." For example, would developers voluntarily adopt practices that would help Hipikat be more useful, such as summarizing and highlighting important parts of discussions in order to make them more understandable if they were recommended by Hipikat to a newcomer in the future? An intriguing question is whether developers would be willing to accept being asked to do more in order to make Hipikat more effective, if they came to recognize the tool's usefulness. (For example, would good development norms, such as requiring bug id's in code check-in comments, be followed to a greater degree?) If developers would do more, a feedback mechanism on the relevancy of Hipikat recommendations might be introduced into the development process. Recommendations could be evaluated along with new code during code review, similarly to the process proposed by Terveen et al. [8].

### 7.5 Hipikat's Applicability

In this research, we focused on a particular subset of virtual software development teams: open-source software (OSS).

An important aspect of OSS is its culture of openness, where all important project discussions are either conducted or summarized in public forums, such as developers' mailing lists. At the same time, open-source software developers rarely keep thorough and up-to-date documentation. The combination of the "culture of communication" and the general disregard for documentation makes open-source software projects a natural fit for a project memory approach.

Within the open-source software community, we expect Hipikat to be most useful to projects that have relatively large developer teams (with 10 or more members) and that have been running for a longer period of time. A project's size encourages the richness and wider range of information sources, while a project's age ensures that there is a sufficient accumulation of experiences to go into the project memory; we expect Hipikat will become increasingly useful as time passes and the project grows and evolves. Given these likely preconditions for a successful introduction of Hipikat into a project, we believe it would be most useful in the following types of situations:

**Refining existing functionality**. The functionality to be added in the easy task in the Eclipse newcomer study built on top of an existing feature that was added at a discrete point in time and that had a very distinct record in the project memory. These factors makes it relatively easy to find the location for the modification and to see how to graft the functionality onto the existing system from the Hipikat recommendations.

**Learning API usage**. The "moderately useful" case in the recommendation quality study is an example where a developer was implementing a common procedure that required knowing a complex API. Hipikat recommendations identified several instances when this action was implemented elsewhere in the system. From those examples, it was easy to see the relevant API and how it is to be used because of the common elements that reappear in multiple different contexts.

**Recovering design rationale**. One of the top recommendations related to the difficult task in the Eclipse newcomer study provides a good example of recovering design rationale. The recommendation includes a lengthy discussion of several behavioral issues that arose when related functionality was added to another Eclipse module. The same issues would have to be considered during the implementation of the new functionality, although its solution would be implemented somewhat differently because of internal module differences. Nevertheless, the discussion and the choices made in the older bug are relevant, especially for the sake of consistency in behavior in similar components across the entire system.

**Avoiding common errors**. Sometimes there are undocumented aspects of the API that cause problems to developers who are not familiar with it. These are often reported as bugs in the bug-tracking system or mentioned on the discussion lists, where more experienced developers can give an explanation. Although this explanation is not always transferred to formal documentation, it becomes a part of the project memory accessible through Hipikat.

## 8 Summary

Thanks to electronic communication mechanisms, groups today can work with members distributed over various locations and multiple time zones. It can be difficult for newcomers to join such groups because it is hard to obtain effective mentoring. In this paper, we investigated how an implicit group memory from the digital archives of an open source software project could be utilized to facilitate development. Using Hipikat we can form such a group memory, and Hipikat can recommend appropriate parts of the memory to newcomers working on enhancement tasks. We presented two studies of Hipikat's effectiveness in such situations. The recommendation quality study showed that, in most of the cases examined, Hipikat was able to provide a useful pointer to the files involved in the solution of the task, the constructs necessary for the solution, or both. The Eclipse newcomer study showed that newcomers can use the information presented by Hipikat to achieve results comparable in quality and correctness to those of more experienced members of the team. We found difficulties for newcomers in understanding recommended artifacts in the context of the past system and in taking the knowledge forward and applying it to the current context.

The two studies support our claim of Hipikat's usefulness to newcomers engaged in software modification tasks. They also raise some wider questions regarding potential limits of using project memories to learn from the past. The recommendation quality study made it clear that Hipikat's usefulness depends on the amount of information, especially communication, captured in the project's artifacts. Knowing whether and to what extent developers would voluntarily adopt practices that help Hipikat make better recommendations would be valuable for further development of the tool. The Eclipse newcomer study highlighted the problem of too-shallow understandings of recommendations. It would be interesting to know if this is an inherent outcome of learning from the past or if systems such as Hipikat can be designed to encourage deeper understanding.

## Acknowledgments

This work was supported by NSERC and IBM as part of the Consortium for Software Engineering Research in Canada. The New Media Innovation Centre (Vancouver) and the National Research Council Canada's Institute for Information Technology (Ottawa) provided observation facilities for the Eclipse newcomer study. The authors would like to thank the study participants for their time and effort, and the reviewers for their many helpful suggestions that greatly improved the organization of the paper.

## REFERENCES


[1] S.E. Sim and R.C. Holt, "The Ramp-Up Problem In Software Projects: A Case Study of How Software Immigrants Naturalize," *Proc. 20th Int'l Conf. Software Eng.,* pp. 361-370, 1998.

[2] L.M. Berlin, "Beyond Program Understanding: A Look at Programming Expertise in Industry," *Proc. Empirical Studies of Programmers: Fifth Workshop,* pp. 6-25, 1993.

[3] J.D. Herbsleb, A. Mockus, T.A. Finholt, and R.E. Grinter, "An Empirical Study of Global Software Development: Distance and Speed," *Proc. 23rd Int'l Conf. Software Eng.,* pp. 81-90, 2001.

[4] M.S. Ackerman and T.W. Malone, "Answer Garden: A Tool for Growing Organizational Memory," *Proc. Conf. Office Automation Systems,* pp. 31-39, 1990.

[5] L.M. Berlin, R. Jeffries, V.L. O'Day, A. Paepcke, and C. Wharton, "Where Did You Put It? Issues in the Design and Use of a Group Memory," *Proc. SIGCHI Conf. Human Factors in Computing Systems,* pp. 23-30, 1993.

[6] J. Grudin, "Groupware and Social Dynamics: Eight Challenges for Developers," *Comm. ACM,* vol. 37, no. 1, pp. 92-105, Jan. 1994.

[7] D. Čubranić, G.C. Murphy, J. Singer, and K.S. Booth, "Learning From Project History: A Case Study for Software Development," *Proc. ACM Conf. Computer Supported Cooperative Work (CSCW '04),* pp. 82-91, 2004.

[8] L.G. Terveen, P.G. Selfridge, and M.D. Long, "From 'Folklore' to 'Living Design Memory'," *Proc. SIGCHI Conf. Human Factors in Computing Systems,* pp. 15-22, 1993.

[9] R. Lougher and T. Rodden, "Supporting Long Term Collaboration in Software Maintenance," *Proc. Conf. Organizational Computing Systems,* pp. 228-238, 1993.

[10] B.J. Rhodes and T. Starner, "Remembrance Agent: A Continuously Running Automated Information Retrieval System," *Proc. First Int'l Conf. Practical Application of Intelligent Agents and Multi-Agent Technology (PAAM '96),* pp. 487-495, 1996.

[11] Y. Ye and G. Fischer, "Information Delivery in Support of Learning Reusable Software Components on Demand," *Proc. 2002 Int'l Conf. Intelligent User Interfaces (IUI '02),* pp. 159-166, 2002.

[12] P. Pirolli and J. Anderson, "The Role of Learning From Examples in the Acquisition of Recursive Programming Skills," *Canadian J. Psychology,* vol. 35, pp. 240-272, 1985.

[13] M.B. Rosson and J.M. Carroll, "The Reuse of Uses in Smalltalk Programming," *ACM Trans. Computer-Human Interaction,* vol. 3, no. 3, pp. 219-253, 1996.

[14] B.M. Lange and T.G. Moher, "Some Strategies of Reuse in an Object-Oriented Programming Environment," *Proc. SIGCHI Conf. Human Factors in Computing Systems,* pp. 69-73, 1989.

[15] E.S. Raymond, *The New Hacker's Dictionary,* third ed. MIT Press, 1996.

[16] M.B. Rosson, J.M. Carroll, and C. Sweeney, "A View Matcher for Reusing Smalltalk Classes," *Proc. ACM SIGCHI Conf. Human Factors in Computing Systems (CHI '91),* pp. 277-283, 1991.

[17] D.F. Redmiles, "Reducing the Variability of Programmers Performance Through Explained Examples," *Proc. ACM INTERCHI '93 Conf. Human Factors in Computing Systems,* pp. 67-73, 1993.

[18] H. Gall, M. Jazayeri, and J. Krajewski, "CVS Release History Data for Detecting Logical Couplings," *Proc. Int'l Workshop Principles of Software Evolution (IWPSE '03),* pp. 13-23, 2003.

[19] R.E. Grinter, "Using a Configuration Management Tool to Coordinate Software Development," *Proc. Conf. Organizational Computing Systems,* pp. 168-177, 1995.

[20] A. Chen, E. Chou, J. Wong, A.Y. Yao, Q. Zhang, S. Zhang, and A. Michail, "CVSSearch: Searching through Source Code Using CVS Comments," *Proc. Int'l Conf. Software Maintenance (ICSM 2001),* pp. 364-373, 2001.

[21] A. Mockus and J.D. Herbsleb, "Expertise Browser: A Quantitative Approach to Identifying Expertise," *Proc. 24th Int'l Conf. Software Eng. (ICSE '02),* pp. 503-512, 2002.

[22] D.W. McDonald and M.S. Ackerman, "Expertise Recommender: A Flexible Recommendation System and Architecture," *Proc. ACM 2000 Conf. Computer Supported Collaborative Work,* pp. 231-240, 2000.

[23] T. Zimmermann, P. Weisgerber, S. Diehl, and A. Zeller, "Mining Version Histories to Guide Software Changes," *Proc. 26th Int'l Conf. Software Eng. (ICSE '04),* pp. 563-572, 2004.

[24] A.T.T. Ying, G.C. Murphy, R. Ng, and M.C. Chu-Carroll, "Predicting Source Code Changes by Mining Revision History," *IEEE Trans. Software Eng.,* vol. 30, pp. 574-586, Sept. 2004.

[25] P.P. Chen, "The Entity-Relationship Model—Toward a Unified View of Data," *ACM Trans. Database Systems,* vol. 1, no. 1, pp. 9-36, Mar. 1976.

[26] A. Mockus, R.T. Fielding, and J. Herbsleb, "Two Case Studies of Open Source Software Development: Apache and Mozilla," *ACM Trans. Software Eng. and Methodology,* vol. 11, no. 3, pp. 1-38, July 2002.

[27] D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H.F. Nielsen, S. Thatte, and D. Winer, *Simple Object Access Protocol (SOAP) 1.1,* World Wide Web Consortium, 2000.

[28] F.-C. Cheong, *Internet Agents: Spiders, Wanderers, Brokers, and Bots.* Indianapolis, In: New Riders Publishing, 1996.

[29] D.M. German, "Mining CVS Repositories, the Softchange Experience," *Proc. First Int'l Workshop Mining Software Repositories (MSR '04),* pp. 17-21, May 2004.

[30] S. Dumais, "Improving the Retrieval of Information from External Sources," *Behavior Research Methods, Instrument, and Computers,* vol. 23, no. 2, pp. 229-236, 1991.

[31] S.C. Deerwester, S.T. Dumais, T.K. Landauer, G.W. Furnas, and R.A. Harshman, "Indexing by Latent Semantic Analysis," *J. the Am. Soc. of Information Science,* vol. 41, no. 6, pp. 391-407, 1990.

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

[33] D. Čubranić, "Project History as a Group Memory: Learning from the Past," PhD dissertation, Univ. of British Columbia, 2004.

[34] M.A. Hearst, "TextTiling: A Quantitative Approach to Discourse Segmentation," *Computational Linguistics,* vol. 23, no. 1, pp. 33-64, Mar. 1997.

[35] A. Arampatzis, "Adaptive and Temporally-Dependent Document Filtering," PhD dissertation, Katholieke Univ. Nijmegen, Nijmegen, The Netherlands, 2001.

[36] M. Balabanović and Y. Shoham, "Fab: Content-Based Collaborative Recommendation," *Comm. ACM,* vol. 40, no. 3, pp. 66-72, Mar. 1997.

[37] W. Hill, L. Stead, M. Rosenstein, and G. Furnas, "Recommending and Evaluating Choices in a Virtual Community of Use," *Proc. ACM Conf. Human Factors in Computing Systems,* vol. 1, pp. 194-201, 1995.

[38] U. Shardanand and P. Maes, "Social Information Filtering: Algorithms for Automating 'Word of Mouth'," *Proc. ACM Conf. Human Factors in Computing Systems,* vol. 1, pp. 210-217, 1995.

[39] J.A. Konstan, B.N. Miller, D. Maltz, J.L. Herlocker, L.R. Gordon, and J. Riedl, "GroupLens: Applying Collaborative Filtering to Usenet News," *Comm. ACM,* vol. 40, no. 3, pp. 77-87, Mar. 1997.

[40] M. Chalmers, K. Rodden, and D. Brodbeck, "The Order of Things: Activity-Centred Information Access," *Proc. Seventh World Wide Web Conf.,* pp. 359-367, 1998.

[41] R.B. Allen, P. Obry, and M. Littman, "An Interface for Navigating Clustered Document Sets Returned by Queries," *Proc. Conf. Organizational Computing Systems (COOCS '93),* pp. 166-171, 1993.

[42] V. Sabol, W. Kienreich, M. Granitzer, J. Becker, K. Tochtermann, and K. Andrews, "Applications of a Lightweight, Web-Based Retrieval, Clustering, and Visualisation Framework," *Proc. Conf. Practical Aspects of Knowledge Management (PAKM '02),* pp. 359-368, 2002.

[43] C.-M. Chen, N. Stoffel, M. Post, C. Basu, D. Bassu, and C. Behrens, "Telcordia LSI Engine: Implementation and Scalability Issues," *Proc. 11th Int'l Workshop Research Issues in Data Eng. (RIDE '01),* pp. 51-58, 2001.

[44] C. Tang, S. Dwarkadas, and Z. Xu, "On Scaling Latent Semantic Indexing for Large Peer-to-Peer Systems," *Proc. 27th Ann. Int'l Conf. Research and Development in Information Retrieval (SIGIR '04),* pp. 112-121, 2004.

**Davor Čubranić** received the BS degree from the University of Southern Mississippi in 1995 in computer science and mathematics, and the MSc and PhD degrees in computer science from the University of British Columbia in 1998 and 2005, respectively. He is currently a postdoctoral researcher in the Department of Computer Science at the University of Victoria. His research interests are in collaborative software development, computer-supported collaborative work, empirical methods in software engineering, and software engineering education.

**Gail C. Murphy** received the BSc degree in computing science from the University of Alberta in 1987 and the MS and PhD degrees in computer science and engineering from the University of Washington in 1994 and 1996, respectively. From 1987 to 1992, she worked as a software designer in industry. She is currently an associate professor in the Department of Computer Science at the University of British Columbia. Her research interests are in software evolution, software design, and source code analysis. She is a member of the IEEE Computer Society.

**Janice Singer** received the PhD degree in cognitive psychology from the University of Pittsburgh. She is a cognitive psychologist working in the National Research Council of Canada's Software Engineering Group. She also heads the NRC's Human-Computer Interaction program. Her interests lie in collaboration, cognition, and improving software processes and tools by understanding the cognitive and social demands of work.

**Kellogg S. Booth** received the BS degree from Caltech in 1968 in mathematics, and the MA degree in 1970 and the PhD degree in 1975 from the University of California, Berkeley, in computer science. He is a professor of computer science and the founding director of the Media and Graphics Interdisciplinary Centre at the University of British Columbia. He has worked in the fields of computer graphics and human-computer interaction since 1968. Prior to joining UBC, he was a faculty member in the Department of Computer Science at the University of Waterloo (1977-1990), and before that a staff member at Lawrence Livermore National Laboratory (1968-1976). Research interests include human-computer interaction, collaboration technology, visualization, computer graphics, user interface design, and analysis of algorithms. He is involved in a number of interdisciplinary research projects at UBC and elsewhere.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.