Project History as a Group Memory: Learning From the Past

by

Davor Čubranić

M.Sc., University of British Columbia, 1998

B.S., University of Southern Mississippi, 1995

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

Doctor of Philosophy

in

THE FACULTY OF GRADUATE STUDIES

(Department of Computer Science)

The University of British Columbia

December 2004

Abstract

New members of software development teams must come up-to-speed on a large amount of information before becoming productive, even if they have previous software development experience. Often, this knowledge is gained through mentoring: an experienced colleague monitors the newcomer's progress on his or her first assigned tasks, and provides 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 quick questions asked over the cubicle divider or at the water cooler.

However, these light-weight channels are not always available in virtual teams, where the members of the team are not collocated. Moreover, workers are less likely to help their non-collocated colleagues, making it even harder for a newcomer to come up to speed on a project.

The thesis of this dissertation is based on the idea that the collection of all 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 the present needs. I call this implicitly-formed group memory a *project memory* and make three claims: (1) that newcomer software developers can use information from the project memory about past modifications completed on the project to help them effectively perform modification tasks to the system; (2) that the project memory can be built largely automatically, requiring minimal adjustments in work practices of software developers; and (3) that the automatically-built group memory can recommend artifacts useful to the current modification task.

To validate the claims of this thesis, I have developed a project memory model and associated tool, called Hipikat, that recommends relevant artifacts from the memory during a software modification task. This dissertation describes the memory model, the implementation of Hipikat, and its use in a series of case studies to validate the thesis claims.

Contents

List of Tables

List of Figures

Acknowledgements

Every journey must come to an end, and sadly, so does this wild ride called Davor's PhD. It has been a wonderful and truly enriching experience, and the credit for that goes to people who helped me along the way.

First and foremost, I thank my co-supervisors, Gail Murphy and Kellogg Booth. Gail, Kelly: thank you for your guidance, support, insights, and, not the least, patience. You will be an inspiration to me the rest of my career. I hope you forgive me for not deciding through all these years on which side of the software engineering/CSCW fence I sit.

Along with Gail and Kelly, the rest of my supervisory committee helped steer this research through to its completion, and the examining committee nudged me forward in the homestretch. Thanks to the supervisory committee members Janice Singer, David Poole, and Norm Hutchinson, to university examiners Lee Iverson and Ron Rensink, and to the external examiner Andreas Zeller for their constructive comments that greatly improved the quality of this dissertation. Thanks in particular to Janice for her advice on experimental design and for her hospitality and help when I was visiting her at the Institute for Information Technology of the National Research Centre in Ottawa (IIT-NRC) to do the data analysis of the Eclipse study. Janice, I will not fail to include a proper control group in every user study that I conduct from now on.

IIT-NRC also provided space and equipment to conduct part of the Eclipse user study, as did the New Media Interaction Centre in Vancouver (NewMIC). I am grateful to both organizations. I am also grateful to IBM Ottawa lab, and especially Marcellus Mindel and Jonathan Arthorne, for letting me be part of their "coop Eclipse bootcamp," technical advice on developing Eclipse plug-ins, access to Eclipse developers, and getting me in touch with coop students who then took part in the Eclipse study. My deepest thanks go to all twelve participants in the Eclipse study, as well as the participants in the Avid study.

Some of the first ideas for what eventually became Hipikat were inspired by my two research internships at Fuji-Xerox Palo Alto Lab (FX PAL) in the summers of 1999 and 2000. I thank my research hosts and mentors, Elizabeth Churchill and Jonathan Trevor, for giving me the opportunity to be a part of such an amazing place at a uniquely dynamic time.

I would like to acknowledge the two labs of which I was a member during my years at UBC: Imager, where the very first prototype of Hipikat was written, and the Software Practices Lab (a.k.a. SPL), an incredible group full of interesting people with exciting ideas that I am very sad to leave. Rob and Elisa, SPL's PhD trailblazers, during their time at UBC and since set an example that I try to follow; I benefitted tremendously from my interactions with Martin, and trying to keep

up with him on our route to completing the doctorate helped me on days (or weeks) when I felt like my sails were losing wind; Brian is a good friend, and I hope we will be able to continue our discussions of software engineering and information management practices, politics, and good brew pubs for many years to come. Over the last two years, several undergrad RA's and coop students worked on Hipikat and Bugzilla plugins and made them much better than I ever could have on my own; my thanks to all of them: Kaili, Derek, Leo, Shawn, Tanya, and Eric, in chronological order.

UBC's Computer Science is an amazing department, and the more I hear of graduate students' experiences elsewhere, the more I realize how good we have it here. Among the many great people I met in the department, I am very glad to consider Doug, Dave, and Alex my friends.

My friends outside the department kept me mostly sane and gave me an appreciation of graduate student life in other disciplines: Hash, Joy, and Carrie. I hope one day soon we will all find ourselves established in Vancouver for the long term.

My doctoral studies were financially supported by a University of British Columbia Graduate Fellowship, by NSERC and IBM as part of the Consortium for Software Engineering Research in Canada, and by NSERC's NECTAR research network. I gratefully acknowledge this support.

Just as important was the moral support of my family in Croatia. Tata, mama, and Vera: I miss you guys and should warn you that I haven't yet given up on moving you to Canada.

Despite the fact that for the last eight years almost all of our communication has been by email, Edo is still my best friend, as he has been for many years now. I hope to go back to graduate school one day for that PhD in Economics that was inspired by so many of our discussions, and shake the neo-liberal economic order to its foundations.

I was no longer a resident of Green College by the time I began my PhD, but I don't think one can ever leave that place behind. Its talks, arts performances, dinners, wine tastings, parties, and of course the reading room remain a powerful magnet. Even more importantly, friends I met there—in particular Arnie, Mike, Dale, Xavier, Clive, Randy, and Kate—some of them the very first people I met when I arrived in Vancouver, are still among my closest friends and I hope will remain so the rest of our lives.

It is also thanks to Green College that I met Nicole, the love of my life. Nicole, you make my every day worth living its every second.

And lastly, my thanks to "Azra" and "Zabranjeno Pušenje," whose music fueled my writing the critical portions of this dissertation.

Sehen Sie diese Stadt?

Das ist Walter.

DAVOR ČUBRANIĆ

The University of British Columbia
19 December 2004

Chapter 1

Introduction

One basic reality of software is that any useful non-trivial software system evolves through its lifetime in response to changes in its environment and in the needs of its users [9]. For a large and long-lived system, this evolution may involve the work of dozens, sometimes hundreds of software engineers, spread across years, if not decades. As time passes and changes accumulate, knowledge about the system is often forgotten or lost as developers leave the project team and new ones join.

The loss of knowledge makes further maintenance and evolution of the system more difficult. A software engineer needs to understand the system to change it effectively. Building adequate understanding can be time-consuming for a developer, especially in the case of a large and complex software system. Trying to perform a change with an incorrect or incomplete understanding of the system's original design concepts leads to what Parnas has called *ignorant surgery* [84], often causing degradation in the system's structure, which in turn increases the effort needed for subsequent changes. Not fully understanding other relevant aspects of the system, such as implicit constraints and dependencies in the code or various design decisions, can lead to other problems, such as code inefficiencies or inconsistency in the user interface.

The challenge of building appropriate knowledge is even more difficult for a software developer who joins an existing software development team. A newcomer to a project must come up-to-speed on a large, varied amount of information before becoming productive, even if he or she has previous software development experience. Sim and Holt, for instance, interviewed newcomers to a project and found that they had to learn intricacies of the code, the development processes being used, and the organizational structure surrounding the project, amongst others [110]. In collocated teams, this knowledge is often gained through mentoring: An existing member of the team works closely with the newcomers, looking over their shoulders and imparting the oral tradition of the project, as the newcomers work on their first assigned tasks [110, 29].

As Berlin observed in her study of interactions between newcomers and mentors [10], mentors use these exchanges to provide a rich array of information, often tangential to the newcomer's actual question, but nonetheless crucial for their development into an expert. 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,

despite having access to the source code: the unwritten design choices, historical quirks, or the code's assumptions and interactions between different modules. Mentors also introduce the newcomer to useful information sources: other teammates' areas of expertise, relevant documentation and its reliability, etc.

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 quick questions asked over the cubicle divider or at the water cooler during chance encounters.

Unfortunately, these light-weight 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 non-collocated colleagues [47], making it even harder for a newcomer to come up to speed on a project in a virtual team.

The hypotheses of this research are premised on the idea that the collection of all 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 the present needs [1, 11]. We call this implicitly-formed group memory a *project memory*.[1] We make three claims:

1. Newcomer software developers can use information from the project memory about past modifications completed on the project to help them effectively perform modification tasks to the system;

2. The project memory can be built largely automatically, requiring minimal adjustments in current work practices of software developers;

3. The automatically-built group memory can select and recommend useful artifacts—in particular past modifications—that identify target classes, reusable code, or other information pertinent to the current change task.

To validate these claims, we have developed a software tool called Hipikat[2] that provides developers efficient and effective access to a project memory. This dissertation describes the design and implementation of this tool and evaluation of its use in three studies conducted to validate the thesis claims.

[1] By "implicitly-formed" we do not mean that this memory contains what Polanyi called tacit knowledge, which is knowledge that is not ordinarily consciously accessible (or expressible in language). Neither does it mean that the implicitly-formed group memory contains *implicit memory*, which is characterized by lack of conscious awareness in the act of recollection [103]. Rather, we mean that the project memory is not created explicitly, because even though the artifacts that form have been formally recorded by the developers, they are written for different purposes.

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

1.1 Difficulties in learning without a mentor

An extreme case of virtual teams in software development is found in open-source projects, which typically accept source code contributions from anyone on the Internet. With such a low barrier to participation, the ratio of newcomers to experienced team members is usually very high, making effective mentorship even more difficult to obtain. Instead, the newcomers are often told simply to "RTSL" ("read the source [code], Luke") or "RTFM" ("read the fine manual"). While sometimes helpful, this rarely substitutes for a real mentor.

1.1.1 Techniques for improving understanding of the source code

Realistically, a programmer does not build his or her understanding of a large software system by simply reading the source code, but rather just the "relevant" parts (that is, by employing what Littman et al. [67] call an *as-needed* strategy and Singer et al. [111] call *just-in-time comprehension*). Unfortunately, when faced with a system containing hundreds of thousands of lines of code, it may be difficult for a newcomer to know where to even *start* exploring the code, and it may be equally difficult to know when to stop.

There exist a wide variety of *source-based* tools, that summarize, visualize, and otherwise present the information from the source code to help the programmer to explore and better understand the code.

Program databases

Static properties of the source code—such as module and function names and references between functions—can be automatically extracted to build a program database. The programmer can use such a database to discover or explore relationships in the code, for example, to locate all callers of a certain function. Some thirty years after Interlisp [115] fully a integrated program database into its development environment, program databases have finally become an essential feature of mainstream IDE's.

The typical user interface to a program database is text-oriented, although there are tools, like the CIA system [22], that can present the results in a graphical form for a better overview of complex relationships. Regardless of these variations in the interface, the basic interaction mechanism between a user and a program database is organized around single queries on individual program elements—for example, all uses of a variable. Such an interface is not well-suited for gaining an overview of a large system because it operates at a very fine granularity of detail, and it is hard for humans to join together multiple probe-like queries into a coherent picture of overall relationships and organization of the system. Such an interaction mechanism means that program databases are most useful for focused exploration of a relatively small section of the code, but makes them inappropriate for use by a newcomer who is trying to determine where to start exploring the code as part of a task.

Visualizing the source code

A database of static properties extracted from the source code does not have to be accessed only through an interface based on queries on individual program elements. The same database can be used to build graphical visualizations of a program, with the motivation that such graphical presentations will be more appropriate to give an overview of the organization of a software system and relationships between its components. Because of the volume of information that needs to be displayed, most techniques apply some form of abstraction to simplify the graphs that are produced when analyzing a program. These abstractions typically take advantage of the hierarchical structure inherent within the programming language, such as packages, classes, and methods in the case of Java, or even the organization of the source code into files and directories. For example, Shrimp [113] uses a zoomable interface where the high-level, "zoomed-out" view can simply represent all files that are in the same directory with a single node in the graph. The tool starts out in the zoomed-in state and allows the user to interactively cluster and abstract away portions of the graph based on a variety of criteria. Unfortunately, it is easy for a newcomer to be overwhelmed by the sheer scale of the initial graph, and it is difficult to know the abstraction criteria that will help him or her gain an understanding relevant to the current task.

Some visualization tools apply automated clustering techniques to identify higher-level abstractions without requiring input from the user. Criteria used to drive the clustering can vary widely. For example, Hutchens and Basili [53] clustered procedures into subsystems based on the amount of code cohesion between them. Merlo et al. [75] used concepts referred to in the comments and function names while Maletic and Marcus [69] combined structural and textual similarity to determine clusters. However, automated techniques typically produce only a limited range of abstractions to organize the program visualization and present an overview of its structure, and the available methods may not always be suitable to the given task or the type of understanding that the newcomer is trying to build.

1.1.2 Techniques for improving program documentation

Program documentation comes in many shapes and forms and can include everything from user manuals to design specifications to low-level implementation details, such as test plans and notes on algorithms used in the code.

If reading the source of a large program is hardly an appealing proposition, reading the manuals is even less so, beyond getting software installed and running. Almost universally, program documentation has a reputation for poor reliability. As Lethbridge et al. found in their study of programmers' use of documentation [64], this bad reputation is borne out in practice. Documentation is frequently out of date and often poorly written. It is no surprise then that software engineers do not trust a considerable fraction of documentation and do not bother consulting it.

Improving the usefulness and maintainability of the documentation has long been a topic of software engineering research.[3] We present here an overview of relevant work in this area grouped into

[3]Usefulness and maintainability often go hand-in-hand, as we have seen, because out-of-date documentation is usually not very useful.

three categories: integrating documentation with the text of the source code, providing hypertext-style documentation, and capturing the rationale behind program design decisions.

Integrating documentation with the source code

In practice, short in-line comments are easy to maintain and programmers often "find them good enough to greatly assist detailed maintenance work" [64]. A number of approaches build on this fact and propose to improve program documentation by expanding the amount of information that is written inside the source code, to the point of blurring the distinction between the documentation and the source code.

There are two general categories of approaches to integrating documentation with the source code, although the exact boundary between the two can be a matter of opinion: (1) developers can follow *coding conventions* such as naming program identifiers according to particular rules or the placement and organization of in-code comments; or (2) the developers can use tools that allow them to mix typographic-quality documentation and the source code within a single document, as in literate programming approaches.

Most large software organizations over time develop their own conventions for writing and formatting the source code, to make it easier to read it, to navigate through it, or to encode additional information. For example, Microsoft developers name variables and functions using the so-called "Hungarian notation," which embeds the data type in the identifier's name. To make it easier for computer tools to find and process documentation located within in-code comments, some coding standards place the comments in a certain location in the code (e.g., the GNU Foundation's style guidelines[4]), or use special tags for markup. The best known of the latter is Sun Microsystem's Javadoc [60], which has become so popular that Javadoc comments are now considered *de rigueur* for all Java code. Similar tools have been written for programming languages other than Java (for example Doxygen[5]). Javadoc, however, was explicitly designed to be used for API specification documentation, rather than application developer documentation [60, page 147]. It is oriented to low-level documentation (written at the level of methods or individual classes), and is therefore of limited help to the newcomer trying to build a more general understanding of the code.

Literate programming [58] is the combination of documentation and source together in a fashion that supports, even encourages, reading by humans. In literate programming systems (for example, Knuth's original `WEB`), documentation and source code are written interspersed in a single file, from which literate programming tools can produce either readable, typographic-quality documentation or compilable program source. Knuth had set out to create a way to turn programs into "*works of literature*," focusing on "explaining to *human beings* what we want a computer to do" [58, p 97, emphasis in the original]. Literate programming systems support this goal through features such as allowing the program to be written in a flexible order of elaboration (that is, independent of that required by the compiler), automatic support for browsing (e.g., table of contents, index, and cross references), and high-quality typeset output (typically using LaTeX). Literate programming has not

[4] `http://www.gnu.org/prep/standards.html`
[5] `http://www.doxygen.org`

proven itself on larger programs or group projects.

Perhaps a more fundamental problem is that most works of literature are written for linear reading, from start to finish, and from a single viewpoint. This approach may not work when writing the documentation of a large software system. Here, a programmer needs information, at different levels of abstraction, on those aspects of the software that are relevant to his or her current task. A literate program, on the other hand, provides only a single flow of explanation, which may or may not be the right one for the task. Therefore, while having an index might help the newcomer decide where to start reading, finding all the necessary information would require a lot of flipping back and forth, following cross-references, and backtracking through the literate program. This is especially the case when concepts span multiple locations in the source code (as in Soloway's *delocalized plans* [112]), something that is difficult to express in a literate programming style.

Hypertext-style documentation

In interviews Lethbridge et al. conducted, software engineers also complained that it can be so difficult to find useful content in documentation that they often do not even bother to look for it. To improve the organization of the documentation and make it easier to find relevant content, hypertext-style documentation systems have been used. Linking and browsing capabilities of hypertext make it a natural choice of a user interface to program databases. Such *hypercode* systems offer interactive browsing of the source code with cross-references represented as hyperlinks (e.g., as in CHIME [32] or the LXR system that is becoming increasingly popular in open-source projects [77]). More importantly, hypertext allows linking of documents at multiple levels of abstraction and reading from different points of view. For instance, in systems like SODOS [50] or ISHYS [40], developers can create "webs" of documents created at all stages of the software lifecycle, from the requirements and specification to architecture and so on, all the way to the source code. Hypertext can also, in principle, accommodate documenting delocalized plans: Soloway and colleagues' proposal for solving this problem—paper documentation where source code is presented in parallel with pointers linking the code to other relevant sections of the program and detailing the rationale for different design and implementation decisions—sounds very much like a vision of hypertext.

Despite potential advantages, hypertext-style documentation has one important drawback: it is time-consuming to write it well, and fiendishly difficult to maintain it. Links have to be created foreseeing future navigation needs, and then kept in sync with the documents as they evolve. Link maintenance does not just mean that the writer has to prevent links from "pointing to nothingness," but also that the navigation paths they form still make logical sense, taking into account the changing content of the documents. Given the difficulties with writing even the plain-text documentation and keeping it up to date, it is not surprising that true hypertext-style documentation has not been widely adopted.

Several approaches have attempted to generate links between source code and documentation automatically, using AI techniques. Statistically-based techniques require minimal additional effort from the users. For instance, Antoniol et al. [4] built a word model for each item of documentation (in their case, a `man` page). To determine the links to the source code, a Bayesian classifier is

run on names of identifiers from a source file for all documentation word models to determine which document is most closely "resembled" by the code in the source file. Similarly, Marcus and Maletic [70] used Latent Semantic Indexing to find which source files were textually closest to a piece of documentation. However, links created by both of these examples were for fairly low-level documentation, essentially a programmer's reference manual. They have not been tested on higher-level documentation which is more abstract and ambiguous.

Design rationale

An important ingredient in understanding the code is knowing the reason why the code is there. This information is not typically captured in the source code for a system, nor can it be recovered by visualization tools. It is also often neglected in traditional documentation, which tends to focus on the "what" and "how," rather than "why."

Design rationale (DR) approaches explicitly aim to articulate and represent the reasons and the reasoning processes behind the design and specification of artifacts [18]. Lee and Lai [63] claim that design rationale can be used to answer questions such as "How did other people deal with this problem?" and "What can we learn from the past ... cases?"

Design rationale originated as a technique in policy planning for developing solutions to the so-called "wicked" design problems, where there are no "right" and "wrong" answers, but rather only degrees of "better" or "worse" given the existing constraints and trade-offs. The core idea was that such problems should be approached through an open-ended dialogue of collaboratively defining and debating issues and alternatives [95]. A number of techniques have subsequently been developed that implement this "argumentative-style" approach, which all use some form of a semi-formal graphical notation that expresses the ideas about the task as nodes linked with relations (e.g., [62, 109]).

These general-purpose design rationale techniques have been applied to the design of software systems (e.g., [26]); other DR systems, following the same basic idea, have been designed specifically for software development [88]. However, design rationale has had limited success gaining acceptance in the software engineering community. Even the empirical studies of DR systems' use and effects on software development have been fairly rare, and the results of those studies have been inconclusive about the benefits of using DR. While there are reported instances of recorded design argumentation that assisted ongoing design work and of access to the record of past decisions being valuable (e.g., [16, 74, 108]), there are also cases where creating the DR structure impeded the "real" work (e.g., instances in [74, 108]). Negative effects have also been observed in other domains when DR was applied [37, 55] including:

- Problems having to divide the knowledge into small discrete chunks and classify them according to the model as it is entered into the system. Sometimes ideas are too-tightly intertwined to be broken up, and even in the best model there will be some information that does not quite fit it, in which case it may easily fall "between the cracks." Also, some stages of the design work may not fit well into the argumentative model of design rationale.

- Spending time on detailed DR analysis of peripheral issues where it did not lead to any useful insight or benefit the project, while taking effort away from more important work.
- Difficulty integrating new contributions, because for the model to be useful and consistent, all relevant contexts to which new nodes could be linked need to be discovered.
- Difficulty understanding the rationale at a later time, even by the person who created it. This happens when some context that was highly relevant at the time was not included in the rationale, perhaps because of its ordinariness or apparent obviousness.
- Capturing every aspect of an issue to make the argumentation clearer, which causes annoyance when those seem to be self-evident at the time of capture or later review.

Given the time constraints facing software developers and the requirement to learn a whole new graphical language for expressing the reasoning behind design decisions, as well as the extra effort required to record those decisions as they are made, such approaches are unlikely to encounter wider adoption by the developers until their effectiveness can be clearly proven, a typical "Catch-22" problem that often arises in software engineering.

1.1.3 Programming from examples

One of the important coding strategies used by both beginner and expert programmers is to use existing code as a template while developing a solution to the task at hand (see Pirolli and Anderson [86] for an example of novices learning programming techniques from code examples, and Rosson and Carroll [97] and Lange and Moher [61] for studies of code reuse by experts).

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 "read the source, Luke": the system's source code is, in effect, full of examples of how to access the API, handle errors, and implement various functionality. Given the scarcity of good documentation and available mentors, developers in open-source software projects rely on such "examples of usage" from the project's source code as perhaps the most useful and trusted source of information. However, as already discussed in Section 1.1.1, 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 that source code, since finding (and recognizing) them requires a certain level of understanding of the code. Thus once the importance of examples in software problem solving was recognized, a number of systems were proposed to help developers find and understand examples relevant to their current problem.

The first system for example-based programming, by Neal [81], presented examples simply as a list in a dialog, without any explanations beyond a one-line description, and with no special search mechanisms. Subsequent systems, such as Rosson and Carroll's *Reuse View Matcher* (RVM) [98], included explanations. RVM organized each example in its collection around a set of scenarios that use a specific class. Each scenario included within its description an animation illustrating the scenario and details of class usage in the scenario (e.g., relationship to other objects in the scenario,

a list of the class's methods relevant in the scenario, and code samples where those methods were invoked). As in Neal's system, RVM's scenarios had to be created and maintained manually.

Redmiles's *Explainer* [89] shifted away somewhat from hand-crafting examples by imposing a uniform model of an example as a semantic network of concepts. The concepts belonged to perspectives (points of view on the example), linked within and across perspectives with typed links. Examples could be shown in multiple views (e.g., code listing, sample execution, and component diagrams), with equivalent concepts in all views highlighted. Text explanations were created by formatting concepts through simple patterns to form sentences. Thus, the author of an example did not have to write every detail by hand, and there was some automated assistance in creating the semantic network (for example, parsing the concepts from the example Lisp code). Ultimately, however, creating examples was still a largely manual process, and there was no assistance for maintaining them as a system evolved.

CodeWeb [76], on the other hand, is entirely automated. The tool applies data mining techniques on a software library and a collection of existing applications that use it to discover "reuse patterns." It presents patterns in the form of association rules, as pairs of library components indicating that application classes that use one component also tend to use the other. However, information provided by CodeWeb is too general to be of real help to a newcomer working on a specific change task. Knowing the likelihood of co-occurrences of class usage and method invocations does not say *when* certain methods should be used—or when they should not—although it may be useful to improve a developer's awareness of the library's API.

1.2 An overview of the Hipikat approach

Maintaining adequate documentation will likely remain a problem for some time to come because developers must choose where to put effort when time is a constrained resource. In such situations, getting the code working is the priority: not everything is written down, and even when it is, it is laconic and not necessarily written at a level appropriate for a newcomer to understand. The same argument applies to the example-based tools described in Section 1.1.3: building and maintaining the examples takes time and effort, and will always be seen as less important than writing code and getting it to run.

Fortunately, the situation is not hopeless. A lot of information that a newcomer typically needs is available in the archives of the mailing lists, the source code versioning system, and the system for recording and tracking work on issues such as problem reports and requested features. It could be argued that the versioning system is an even better source of examples than the current source code because there the changes implementing specific functionality are isolated in discrete revisions, usually with a comment describing the purpose of the change!

However, this information is not easily accessible because of its sheer volume, the lack of tools to search the information effectively, and the difficulty of making connections between logically related items in disparate repositories. General search engines, such as Google,[6] are commonly used for this purpose, but limit the developer to searching for exact words in documents within a

[6] `http://www.google.com`

single collection (i.e., the web). The developer has to know the right terms to use in the search, and the search engine cannot take advantage of the different artifact types and relationships between them which form as a result of the development practices.

In the remainder of this section, we introduce our approach to the problem of building an understanding of a software system by a newcomer developer to the project. We give its practical implementation as a developer tool, called Hipikat, that applies it to a real-world large open source project. We then present an example of using Hipikat on a change task drawn from the development history of a real, large software system.

1.2.1 Overview of implicit project memory approach

This dissertation presents the Hipikat tool. Hipikat is intended to aid a developer working on a change to a software system, with special focus on newcomers to the development team. To help a newcomer in this situation become productive more quickly, Hipikat *recommends* existing artifacts from the history of the project that are relevant to a task that the newcomer is trying to perform. In essence, we consider all of the artifacts that have been produced—the versions of the source, the bugs, archived electronic communication, web documents—as an implicit group memory.

The tool plays two roles. First, the tool infers links between the artifacts that may have been apparent at one time to members of the development team but that were not recorded. These links are determined using heuristics that are to a large extent based on my observations of work conventions and on informal communication with developers in a large open-source software project. The artifacts and the links together form the *project memory*. We call this project memory *implicit* because it is built automatically, monitoring the course of the development, rather than requiring the developers on the project to create it explicitly. Second, using the links, the tool, in a role similar to that of a mentor, suggests possibly relevant parts of the project memory given information about a task a newcomer is trying to perform.

Figure 1.1 shows the schema we use to represent the project memory. There are four types of artifacts represented in the schema: bug and feature descriptions (e.g., items in Bugzilla), which form *change tasks* for the developers to work on; source *file versions* (e.g, checked in a CVS source repository), which implement the changes; messages posted on developer forums (e.g., newsgroups and mailing lists); and other project documents (e.g., design documents posted on the project's web site). These artifacts are created by project members, represented by `Person` in the diagram.

Hipikat works as a client-server system. The client, when commanded by the user, issues a request for suggestions to the server, and displays returned results to the user. The query identifies (anonymously) the user and the artifact for which related items are sought. The server replies with a list of matches that the client then formats and presents in human-readable format.

A developer typically uses Hipikat by performing queries to discover past work done that could be used as an example guiding the current task, or which can provide background information to help better understand the system and the reasons behind decisions.[7] Hipikat becomes another

[7]In the rest of this dissertation, the term "Hipikat user" is equivalent to "a developer using Hipikat to access the project memory."

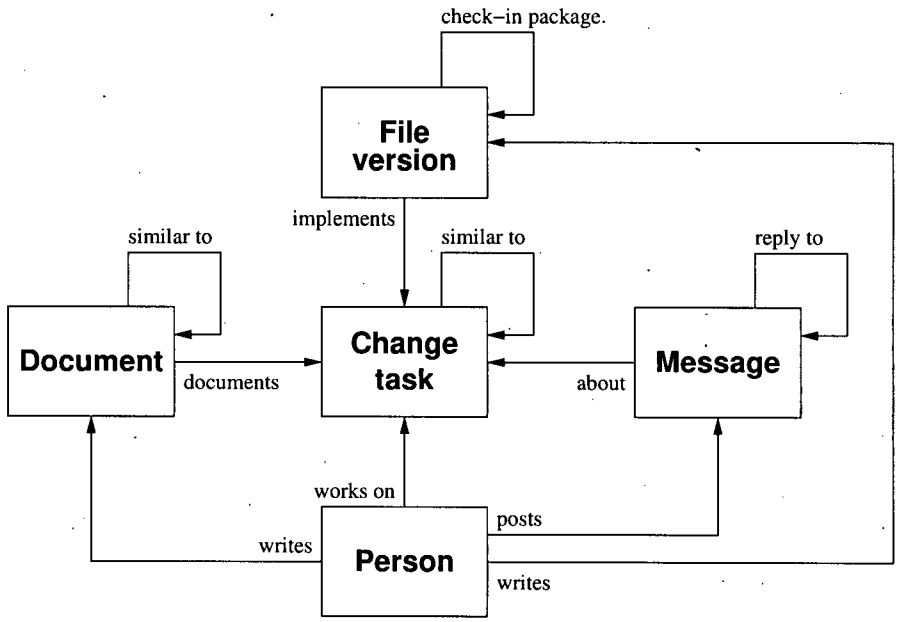

Figure 1.1: Artifact types in the Hipikat project memory and the relationships between them.

tool in the palette available in modern development environments. Recommendations provided by Hipikat can become starting points for further investigation, which in turn can inspire other queries, and so on.

1.2.2 An introduction to the Hipikat tool

To illustrate a typical session using Hipikat, we present a sketch of its use by a software developer in a representative source change task. A detailed description of Hipikat's implementation and functionality is included in Chapter 3; the example shown here is meant to give the reader a flavour of what using Hipikat is like and make descriptions and discussion in the rest of the dissertation more concrete.

In this sketch, we will follow a fictional software developer who is a relatively new member of a software project as he is working on a software modification. We are looking at a situation common in modern software development in which the developer has a written description of the desired functionality, in this case drawn from a database of bug reports and feature requests submitted by users. The developer has already had some experience working on the project, but not enough to be familiar with all aspects of the source code.

The setting

Our example developer is working on the Eclipse project.[8] Eclipse is an integrated development environment. It is written in Java and consists of over 1.9 million lines of source code, distributed over more than 9,000 Java files. Eclipse is developed in an open-source manner (see Appendix A for a brief summary of the open-source approach to software development), and its extensible architecture means that a variety of third-party "plug-ins" have appeared that implement features and tools for a wide range of programming languages and development techniques. Figure 1.2 shows the main Eclipse application window with the Java development environment active.

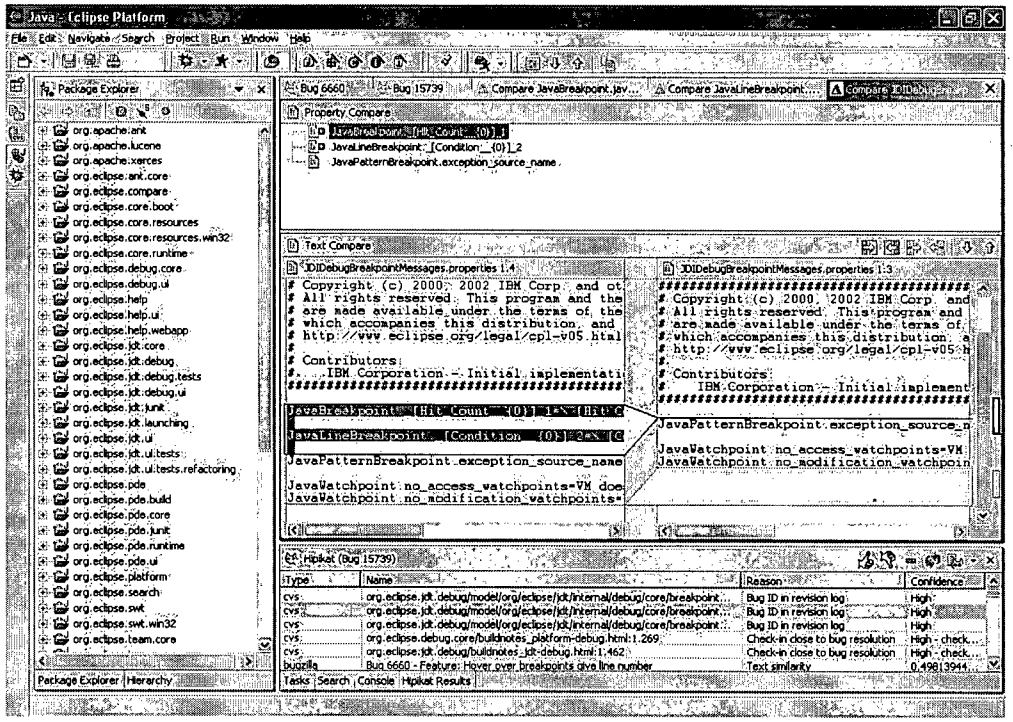

Figure 1.2: The Eclipse integrated development environment.

The task

The base Eclipse distribution offers a rich set of features for editing and debugging Java applications. Breakpoints can be set simply by double-clicking beside the desired source line in the editor.[9] The Eclipse user can set conditions which are evaluated when the execution reaches the breakpoint, and the execution will be suspended only if the condition evaluates to true. The condition can be set

[8] `www.eclipse.org`. We use version 2.0 in this example.

[9] A breakpoint is a debugging facility that suspends the execution of the program at a certain location in the code, enabling the developer to investigate the program's internal state.

from the breakpoint properties dialog (see Figure 1.3) and includes a boolean Java expression or the number of times the breakpoint has been reached.

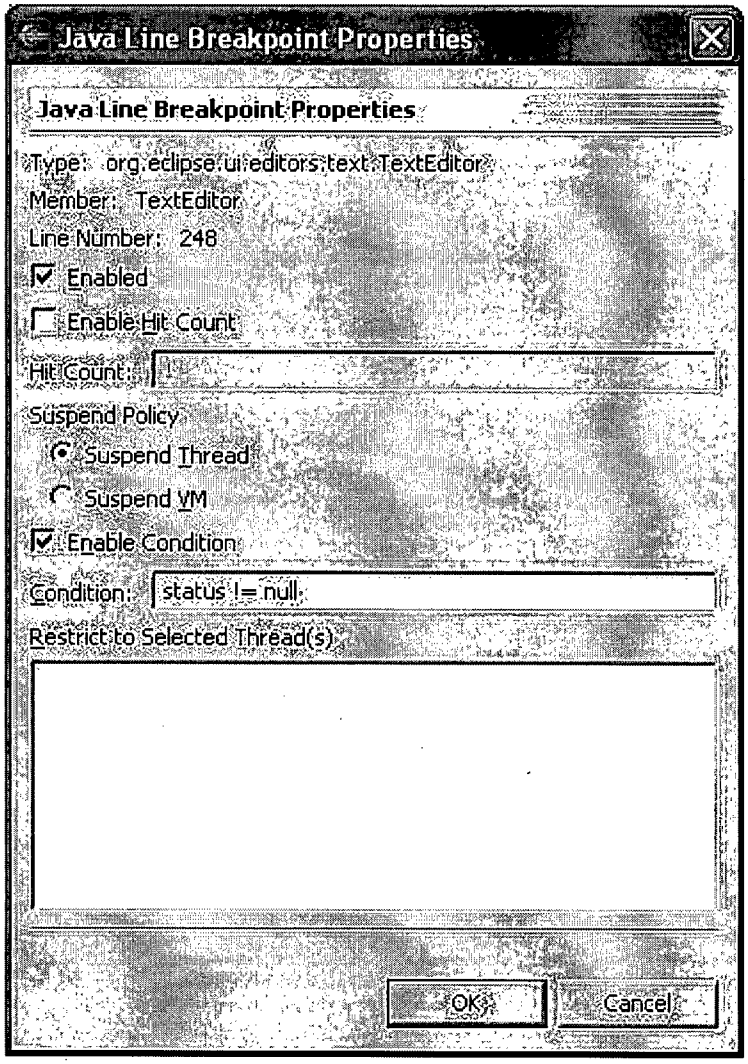

Figure 1.3: Breakpoint properties dialog.

In this scenario, our developer has been asked to modify the feature that displays a popup window when mouse pointer hovers over a breakpoint indicator in the Java program editor. In version 2.0 of Eclipse, this popup appears only for conditional breakpoints, and displays the line number and text of the breakpoint's condition (see Figure 1.4). The modification will introduce hover popup over all breakpoints, even when a breakpoint is unconditional.[10]

[10]This is a real request that was implemented for version 2.1 of Eclipse. The request can be seen at the following URL: `https://bugs.eclipse.org/bugs/show_bug.cgi?id=6660`.

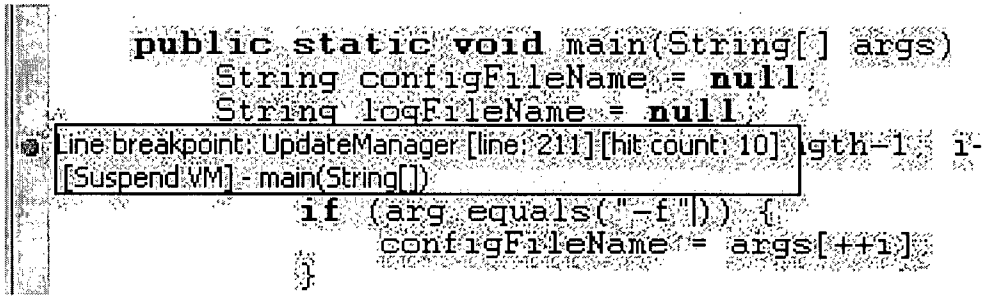

Figure 1.4: Breakpoint hover popup.

A session with Hipikat

The developer starts work by opening the request in the IDE (figure 1.5). He has not worked in this part of the system before, but is aware that the requested feature is a modification of the existing functionality in Eclipse, and concludes that finding where the current behaviour is implemented would be a good starting point. However, the code is large, and it is not obvious how hover pop-ups are implemented. Unsure of what to do, the developer decides to query Hipikat for artifacts related to his task. He right-clicks in the request and selects "Query Hipikat" from the context menu (Figure 1.6).

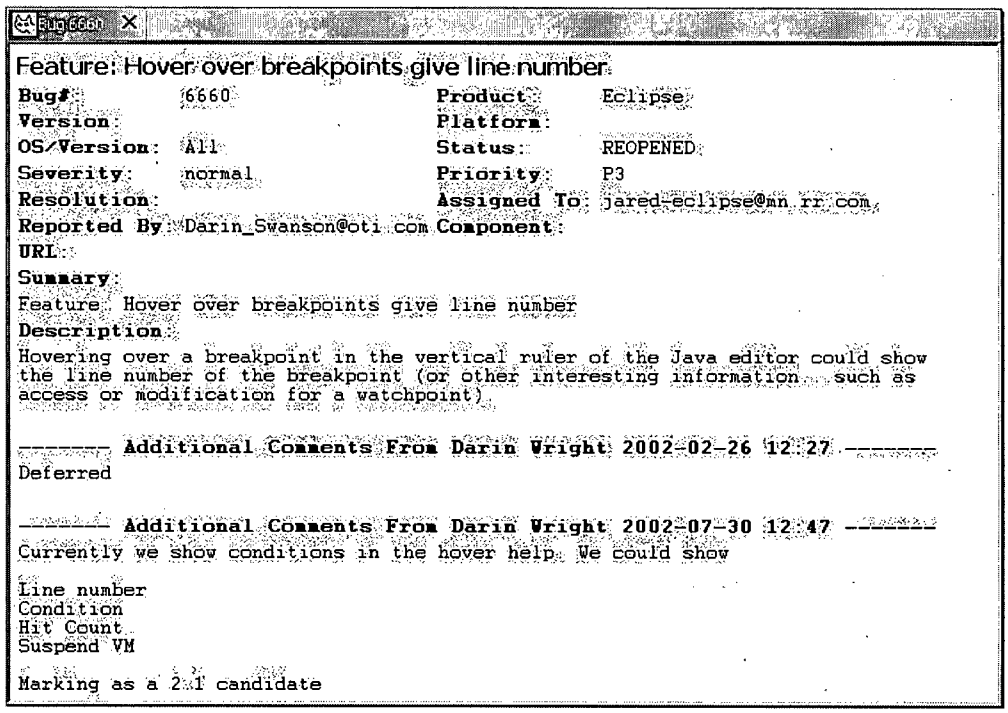

Figure 1.5: The feature request for the example task.

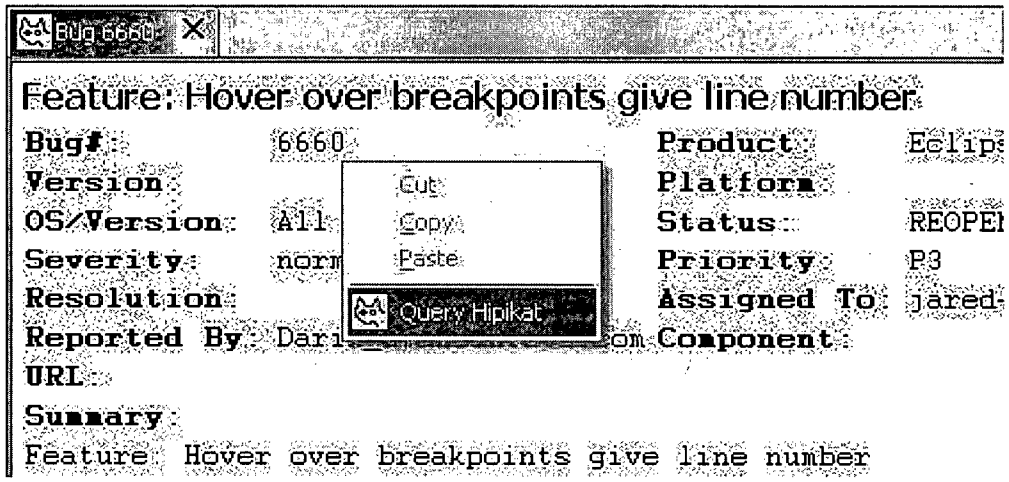

Figure 1.6: Querying Hipikat on the feature request.

In response, the Hipikat server sends a list of recommended artifacts from the project memory, which the client displays in a "Hipikat results" view (see Figure 1.7). At the top of the list are several existing problem reports that were recommended for their textual similarity to the assigned task. The top recommendation sounds like it is the change task which implemented the current hover popup functionality for conditional breakpoints. The developer decides to investigate it more closely and opens the Bugzilla problem report (Figure 1.8).

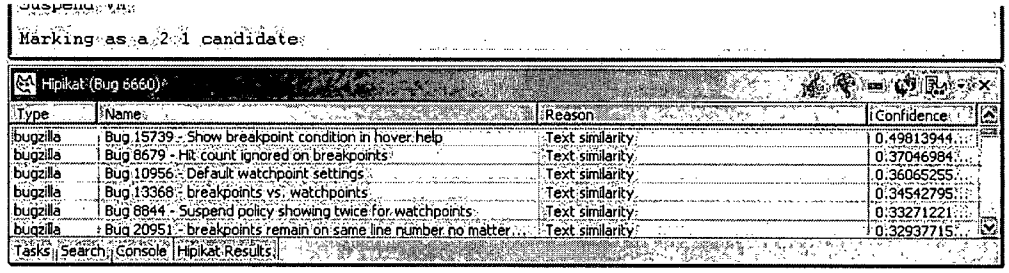

Figure 1.7: Hipikat's recommendations for the starting problem report.

After reading the recommended artifact, the developer feels he is on the right track and decides to query Hipikat on that report to see how the fix was implemented. The list of recommendations returned by Hipikat contains several file revisions that are marked with high confidence for their relevance (see Figure 1.9). There are two .java files—named "JavaBreakpoint" and "JavaLineBreakpoint"—and a .properties file. The developer opens the file version artifacts to see their contents. These are displayed in a "diff" view, which highlights the text that changed compared to the preceding revision (Figure 1.10).

The developer can see that code introduced in these versions deals with setting of "attributes" in "markers." (The developer can now use the IDE to call up the Javadoc for the `Marker` class: it

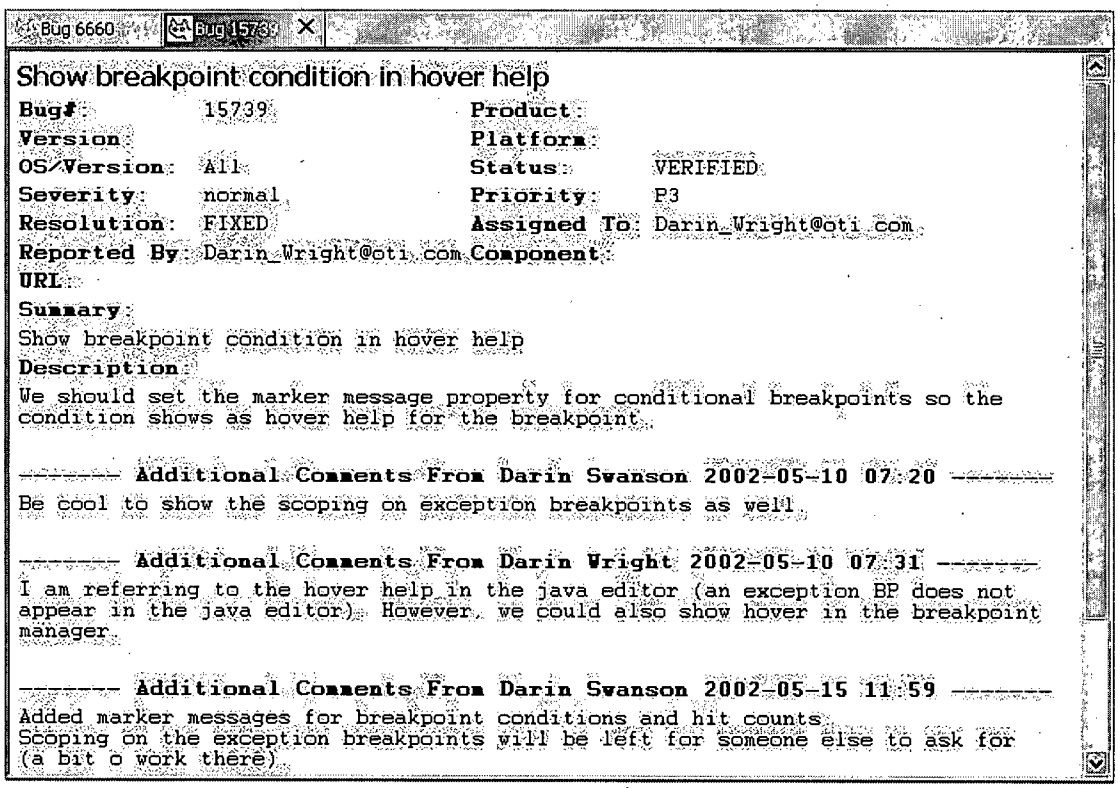

Figure 1.8: Related problem report recommended by Hipikat.

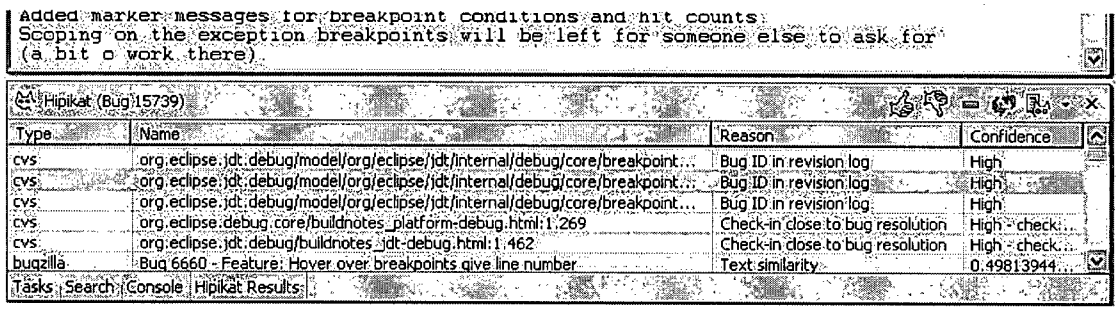

Figure 1.9: Hipikat's recommendations for the related problem report.

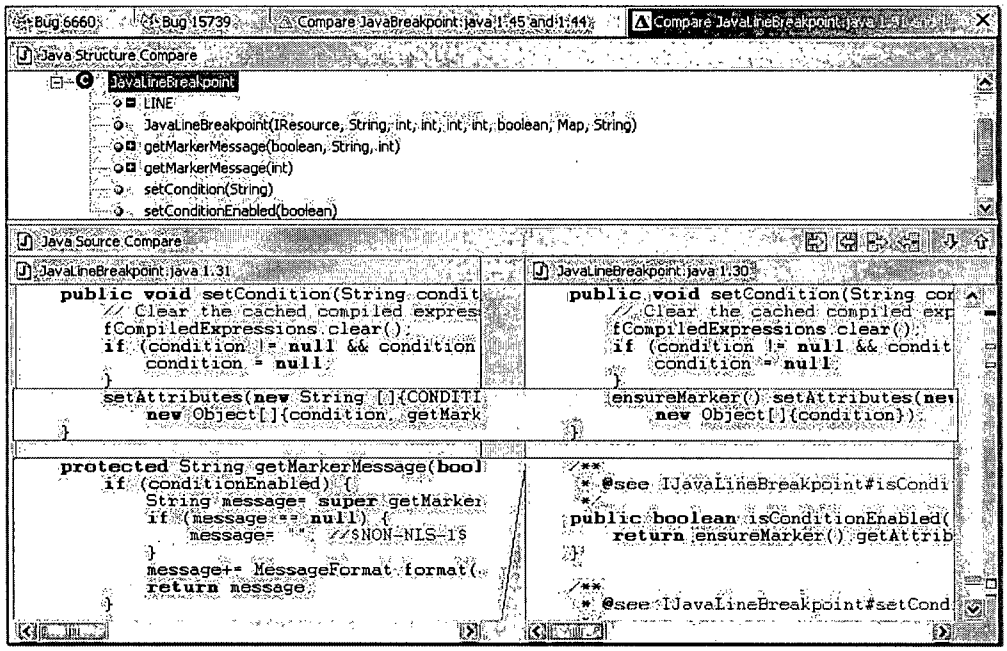

Figure 1.10: Viewing a CVS recommendation. JavaLineBreakpoint revision that implemented the fix to problem report 15739.

is a holder for a set of key-value pairs of named attributes, associated with a "resource" in the IDE.) Apparently, one of the attributes, named `IMarker.message`, is set to what looks like the text that shows up in the hover popup. However, it is not clear how this attribute ends up in the hover, since none of the code recommended by Hipikat seems to have anything to do with the user interface. The developer also notices that this attribute is being set only in the constructor and the setters for the breakpoint's properties (e.g., `setCondition`). It is quite likely then that once this attribute is set, it is the value that will show up in the hover popup, through some still unknown mechanism.

At this point, the developer can test the hypothesis that markers are used to display popup hovers in a practical way by hardcoding the `IMarker.message` attribute to some distinct text and running the modified code to see whether the text appears in the hover popup. The developer tries this approach, and the popup indeed shows the hardcoded text. It looks like the hypothesis is correct. The developer can now try implementing the requested feature, even though he does not understand fully how the UI system detects and displays a hover. All he needs to do on his end is work with the attributes of the breakpoint's marker. Alternatively, he can try using the IDE's cross-referencing capabilities to see which classes access the `IMarker.message` attribute of markers, and follow the chain to the UI classes.

To summarize, even when a modification is relatively simple to implement, such as this example from Eclipse of displaying a hover pop-up with breakpoint properties, it may require understanding of many different subsystems and how they interoperate (in this example, the breakpoints model, markers in an editor, and the graphical user interface). Newcomers to a software project have to

find and understand a large amount of information before they can become effective, which takes time and effort. But with access to Hipikat recommendations from the project memory, a newcomer can see examples of past modifications that can be used as a starting point for his task. These examples will contain the relevant code constructs, show the API usage, sometimes even discuss design alternatives, trade-offs, and caveats. In this example given in this section, the modification was almost trivial once Hipikat recommendations were seen. Still, Hipikat did not serve the solution on a platter, and rarely will. Rather, Hipikat recommendations are there helping newcomers along on the road which they still need to walk themselves as they make the transition from newcomers to experienced members of a software project.

1.3 Summary

This chapter gave a brief overview of drawbacks of existing approaches to improving understanding of a software system when used by developers new to a project. We then introduced our approach to assist newcomers in such situations—the implicit project memory and the Hipikat tool. We then presented a sample session of using the tool in a software modification task. The following list recapitulates the important points of our approach in general and the Hipikat tool in particular:

- Hipikat unites multiple sources of data used in the project. When those sources are kept in separate archives, it is hard for a user to find logical connections between artifacts of disparate types.
- The project memory is built automatically and from existing sources of information. Team members are not required to do additional work that is not going to directly benefit them, which they are unlikely to do willingly and which is one of the main obstacles to successful adoption of groupware.
- No predefined taxonomy of the project memory is imposed on the users. A single taxonomy of group memory is unlikely to work for all users and be relevant in all tasks. As the project evolves, the taxonomy has to evolve with it, which requires costly maintenance.
- Hipikat works as a recommender, making it easier for a newcomer to navigate around the system and find relevant pieces of information as he or she builds an understanding of the project and assigned tasks.
- The query system is essentially point-and-click. Hipikat's user does not have to learn a complex query language or be intimately familiar with the vocabulary of the project.
- Access to the project memory is integrated into the user's work environment (in this case, the IDE), where it is easily invoked from the user's everyday tools and can become part of the regular work practices.

1.4 Organization of the dissertation

In Chapter 2, we review related work. Chapter 3 describes the project memory model and the implementation of the Hipikat tool. In Chapter 4 we describe the validation of thesis claims. In Chapter 5 we discuss the main issues that arose during the development and evaluation of Hipikat, the trade-offs involved, and compare our choices with existing alternatives. Finally, in Chapter 6 we review the claims of this research, describe the contributions of our work, and outline future avenues of research.

Chapter 2

Related work

In this chapter we discuss work related to Hipikat's implicit project memory approach. Several areas are discussed. First, we describe approaches proposed to help organizations build and use a collective "memory" that allows them to benefit from past experiences (Section 2.1). Next, we discuss approaches aimed at unifying different information sources to help developers find information that will help them understand the source code (Section 2.2). Then, we give an overview of recommender systems and discuss ones that assist users find useful information in situations similar to Hipikat's (Section 2.3). Finally, we consider approaches that mine artifact repositories for history data that can be of use to software developers working in the present (Section 2.4).

2.1 Group/organizational memory

It has been long recognized by researchers that groups and organizations possess a "collective" memory that stays even when individual members of the group have left, and that this memory is an important factor in the success of an organization's responsiveness to the changes and challenges of its environment. Walsh and Ungson have given the most comprehensive theory of the organizational memory, which they define as the "stored information from an organization's history that can be brought to bear on present decisions" [119, page 61]. In their theory, this memory is not centrally stored, but distributed across different retention facilities—from individuals and culture to organizational structures and physical setting.

Walsh and Ungson's theory of organizational memory, while comprehensive, does not consider the effect of information technologies on organizational memory. The unique potential impact of those new technologies on organizational memory and, consequently, decision making was first recognized by Huber [52]. Huber specifically stresses the role of advanced information technologies such as information storage and retrieval systems and knowledge-based systems, in creating more timely, comprehensive, and accurate organizational intelligence and in capturing organization members' expertise.

2.1.1 Memory of experience

One approach to building computer-based organizational memory is for the experienced members to do it by externalizing and collecting their knowledge so that everybody can access it.

Ackerman and Malone's *Answer Garden* was probably the first computer-based system whose purpose was to augment an organization's memory [1]. Answer Garden facilitated development of collective databases of commonly-asked questions which could grow "organically" as new questions arose and were answered. In Answer Garden, users browsed the answer database by traversing a tree of diagnostic questions until they reached an appropriate answer in the leaf node. If they could not find an answer, users could contact experts, whose answers would eventually get incorporated into the database. The second version of the system blurred the dichotomy between users and experts, recognizing that everyone can be expert on some issues and novice on others. Questions in Answer Garden 2 were first forwarded to public forums (such as a newsgroup), and only brought to the attention of a smaller group of experts if left unanswered for certain time. Additionally, collected answers could be collaboratively refined by the user community.

Answer Garden's approach was primarily useful for tasks following predefined or common steps, where there is a relatively small set of "paths" in the question tree and a set of "frequently asked" questions that always seem to come up. Although it was deployed in a variety of settings over the years, its application in software development has been limited to a help system on programming for an X window system user interface toolkit.

Terveen et al. [116] introduced a system for recording and disseminating software design knowledge that is usually not written down, or "folklore" as they call it. Their system, *Design Assistant*, used an interaction approach similar to Answer Garden, guiding the developer through a sequence of decisions about design attributes of a particular feature. At the end of the dialogue, the tool produced advice on using the feature given the developer's choices. An important component of this approach was the adjustment to the development process to ensure maintenance and evolution of the Design Assistant's knowledge base: the tool's advice and interaction transcript were treated as part of the design and included in design review. Conclusions of the design review and experiences fixing software faults were then fed back into the knowledge base.

Design Assistant is among the rare instances of computerized support for organizational memory in software development where published reports exist of some success being adopted in a commercial development (AT&T). However, it was a fairly heavy-weight approach, which required significant effort for creation, maintenance and evolution of the organizational memory, and was therefore suited to a style of software development that places large emphasis on well-defined software process, into which these activities could be incorporated and accepted by both the management and the developers. Hipikat, on the other hand, does not require any change to the development process and records much of the same "folklore" by unobtrusively capturing developers' discussions and actions. Another drawback of the Design Assistant's approach is the "dialogue tree" model of interaction with the user. The model is difficult to create because identifying the design attributes of the target domain is a manual process. Furthermore, complex design decisions often cannot be expressed as a series of "yes/no" questions; the design dialogue is more likely a graph than a tree

because some options may involve trade-offs that affect previous choices. Lastly, some aspects of design include broad concepts that affect wide areas of the system and cannot be solved with computer-generated "recipes."

Carroll et al. [17] proposed an alternative method for lightweight recording and organizing of informal history and rationale that design teams create and share in the course of their work. Their Raison d'Etre system provided access to a database of video clips containing stories and personal perspectives of design team members. The clips could be recorded as the project progressed, giving an overview of the evolution of the design and personal experiences. However, the database was stand-alone; there was no way to integrate it with the source code—for example, for seeing the code as it was when the clip was recorded, or as it evolved afterwards. Also, the videos came from interviews with the developers that Carroll et al. conducted, and had to be laboriously divided into clips and appropriately organized. Therefore, the system was more a proof of concept than a practical solution.

2.1.2 Memory of interactions

An alternative approach to building organizational memories does not require users to explicitly externalize their knowledge, which, as we have argued in the previous section, carries with it a number of difficulties. Instead, these approaches aim to leverage existing artifacts and make it as easy as possible to create the memory. A typical example is the TeamInfo system, created by Berlin et al. [11], which builds its memory from email messages. Berlin et al. call TeamInfo a *group memory*, a variation of organizational memory that is tailored to the needs of a small group of collaborating colleagues. It is more than just an email archive: TeamInfo is intended as a repository of information of long-term value to the group. Berlin et al. reported that users submitted to TeamInfo a variety of items that they thought would be useful to the group in the future—from "cc'ing" TeamInfo on interesting discussions, to forwarding useful nuggets of information received over email or by other means. To make it easier to easier to retrieve information, items in the group memory were classified to one or more group-defined categories, either automatically using pre-configured keyword patterns or explicitly by users. In addition, to promote group awareness of the activity in the group memory, users had the options to receive email notifications of items recently added to TeamInfo.

Hipikat takes the same approach to light-weight creation of group memory from existing artifacts. However, it lowers further the bar on effort needed to build the group memory by monitoring all activity in public forums, rather than requiring its users to explicitly submit items to the group repository. On the other hand, our approach has the drawback that there is no way to collect useful items created outside regular sources of information. For example, an exchange over personal email could easily be accommodated in TeamInfo by forwarding the email afterwards to TeamInfo's address. The equivalent workaround for Hipikat would require that the email be forwarded to one of the information sources that are monitored by Hipikat, such as the mailing list; the difference from TeamInfo is that the forwarded message will add to the list traffic even though its author merely wanted to have it archived in the project memory for later access.

On the related note of minimizing the effort required, Hipikat does not use a taxonomy of categories to organize the collection, but instead recommends relevant items on a case-by-case basis. There is another reason for avoiding predefined taxonomy of the group memory: using categories was actually a significant source of user frustrations and even provoked arguments about the "right" taxonomy and the "right" classification of individual items. As Berlin et al. discovered, there are distinct styles of filing the items into the group memory (similar to the styles of managing email messages in personal folders [122]). Individuals using different filing styles were likely to classify the same items differently, and would also look for them in different categories, making it harder to find information in the memory and diminishing its usefulness. Items that were mutually related could therefore be filed in different categories, losing the connection between them. Given that these problems surfaced even for the small group using TeamInfo, it is likely that they would be even more pronounced, and more difficult to manage, for a large open-source project that we were targeting with Hipikat.

Lastly, and perhaps most significantly, Hipikat goes beyond Berlin et al.'s approach by correlating information from multiple sources, including both the communication and "work" artifacts (e.g., discussion about a bug with the code implementing the solution).

2.2 Unifying information sources

One of the challenges in using design rationale is that the rationale is recorded and evolves separately from the design artifact. This separation causes problems such as difficulty finding the exact rationale that motivated a particular feature of the evolving artifact, not implementing in the artifact the design decisions that were agreed upon, or discussing design issues that are irrelevant to the current state of the artifact. Reeves and Shipman recognized that "discussions *about* the design must be embedded *in* the design" [90, p 394, emphasis in the original]. They implemented these principles in their XNETWORK system, which allowed discussions to be embedded in the artifacts. XNETWORK supported LAN design work, where the design artifact—the network layout—was represented graphically. Shorter discussions were shown as PostIt-like notes in the design; longer, design-rationale argumentation were on separate pages connected to the relevant elements of the network. Its main limitation was that its applicability was constrained to domains that could be represented graphically. While this may be the case for software design, which is commonly represented in just such a way, software ultimately is written in plain text. Also, argumentation in XNETWORK was linked to individual objects in the layout (e.g., a router), so it would be difficult to express argumentation relevant to a higher level of abstraction or touching on multiple parts of the design.

Initial steps towards integrating software artifacts with developers' communication were made by Lougher and Rodden [68]. Their system was conceptually similar to computer tools for software inspection [35]. Tools such as CAIS [71] and HyperCode [85] supported asynchronous code inspection by allowing engineers to annotate the source code under inspection with their comments and distribute them to other participants in the activity. However, although these comments were captured electronically, they were lost once the inspection was over. Lougher and Rodden recog-

nized that these comments could have a longer-lasting value and be useful in future maintenance activities. They developed a system that allowed maintenance engineers to make annotations on the code as they changed it or reviewed their colleagues' changes, capturing rationale and making long-term collaboration possible. However, this approach requires the user to look at the exact spot in the source code to see the annotation, which may not be as useful for a relative newcomer trying to grasp tens of thousands of lines of source. Also, although annotations could be made at different levels in the program's structural hierarchy (e.g., a line or a function), there was no way to capture comments that referred to scattered locations in the code.

Raison d'Etre's idea of recording the developer's views of the project (see Section 2.1.1) was taken one step further by integrating the source code with video and other media, implemented by Chieh et al. in *Variorum* [25]. Variorum supports program documentation in the form of audio, video, and pen drawing annotation of the source code. It is intended for recording software "walk-throughs," however, and there is no support for program evolution or design discussion. Also, it has only been applied on small programs (less than 1,000 lines) and it is unclear how it would support explanations of larger and more complex systems, especially high-level design issues since the explanations are linked to lines of code.

Lindstaedt and Schneider [66] demonstrated a combination of a multimedia software walk-through system, similar to *Variorum*, with an email repository similar to TeamInfo (see Section 2.1.2). Their FOCUS system allows the recording of multiple execution paths through a program, each path demonstrating a different functionality. Paths can have associated with them a developer's explanations (audio, video, or text) and follow-up discussions. These so-called "explanation paths" can intersect, for example at a method that they all include in the explanation. A collection of explanation paths for a given program can be saved as a set of interrelated HTML pages and media files accessible via a web browser. Thus saved, they become part of GIMMe, an online repository similar in functionality to TeamInfo. GIMMe is intended to supplement FOCUS with support for long-term collaboration: developers can locate explanations created in FOCUS by searching or browsing the category hierarchy similar to TeamInfo's. Emails sent as follow-ups to points in the explanation paths and other development-related discussion are all captured in GIMMe and become part of the group memory.

While in some ways similar to Hipikat, Lindstaedt and Schneider's approach was targeting an audience of researchers who create software prototypes as part of their research. It focused on preservation of experiences once the prototype was completed and facilitating transmission of those experiences to future efforts. The explanations were created on a finished piece of software, at the end of a project, and the knowledge that was captured was essentially reflections on the past. There was no support for evolution of the code, or capturing design discussions at the time they were going on, all of which are essential part of Hipikat's philosophy.

2.3 Recommender systems

Recommender systems are programs which attempt to predict items (for example, music or books) that a user may be interested in based on some information about the user's profile [91]. The criteria

used to make a recommendation can be content-based or collaborative [107]. In content-based systems, the decision whether to recommend an item is based solely on its content and its fit with the user's profile. Collaborative recommenders—also known as collaborative filtering systems—are essentially agnostic about the content of the item but instead use other users' ratings of it. Those ratings can be explicit, when users explicitly rate items for their usefulness, or implicit, when ratings are inferred from users' actions (e.g., buying a book is an implicit positive recommendation which an online bookstore can use to recommend it to other customers with similar interests).

Currently, Hipikat's recommendations are content-based, although it uses some collaborative-based elements in the way items are ordered in a recommendation list (see Section 3.1.2 for details). (For a discussion of extending this recommendation model, see Section 5.1.3.)

Similarly to Hipikat, Ye and Fischer's *CodeBroker* [123] uses content-based criteria to determine software artifacts to suggest in the context of a developer's current task. However, CodeBroker is tailored to helping a developer on small-scale reuse tasks: it monitors a developer's use of a text editor watching for the method declarations and the descriptions of those methods in Javadoc comments, and uses that information as a query to a library to find potential components that could be reused instead of a new component being created. In contrast, when used as a reuse tool, Hipikat works at the granularity of a task, providing such information as documents describing how a component is to be used with other components. The CodeBroker approach also relies on the developer's 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, making use of information that is potentially more informal.

In this regard, Hipikat is more similar to the *Remembrance Agent* [92], which mines existing, non-structured collections, such as user's email folders, to present documents relevant to the one currently being edited. Like Hipikat, Remembrance Agent imposes no additional work on the user, such as manually annotating the collection before it can be used. Its database is built automatically, by indexing local directories of text files, email messages and other documents such as LaTeXor HTML files, or bibliographies. Unlike Hipikat, however, Remembrance Agent continually queries its database, based on the text surrounding the current cursor position in the editor. Also, because it is a general "personalized information manager," Remembrance Agent does not have any structure within its document collection, and the only criteria for recommending an artifact is its textual similarity to the current editing context.

Letizia [65] is a content-based recommender that is intended to make exploring complex information spaces more efficient. Letizia assists users in web browsing by automatically recommending pages in the neighbourhood of the current page that the user might want to visit. The criteria for recommendation are based on the current "interest profile," created from contents of the pages recently viewed by the user. Letizia then acts as an "advance scout," following links from the user's current web page and bringing up those pages in the "local neighbourhood" that match the user's interest profile.

Like Remembrance Agent and unlike Hipikat, Letizia makes its recommendations automatically, every time a web page is visited. Like Hipikat (and unlike Remembrance Agent), it can exploit the structure inherent in the domain—in this case, the page links to explore the "neighbourhood."

Letizia also incorporates a temporal component in its recommendation algorithm: the interest profile is cumulatively built from the user's activity over time. (For a discussion of extending Hipikat to incorporate a temporal awareness of user's activity, see Section 5.1.3.) However, Letizia works only within a collection of a single document type—the web of HTML pages—and within a very small subset of it at that: pages within a few links' distance from the current web page.[1] It is intended to help a user avoid wasting time on exploring irrelevant parts of a web site, not discovering related documents within and across multiple information sources.

Fagrell's *Newsmate* [36] applies many of the same principles behind Hipikat to the journalism domain. It is a recommender that uses a journalist's "todo" list (kept on a PDA) to search an archive of news stories and recommend stories related to entries in the "to-do" list, and people who have written stories on those topics in the past. Newsmate can also search other "todo" lists to detect duplication of effort when two journalists work on same or similar stories. Like Hipikat, Newsmate builds its database from artifacts that are created in the normal course of work, which for journalists are news articles. It uses an electronic information organizer, a To-Do list kept on a PDA to infer the task context and search the database for artifacts. Although it does not provide access to past communication like Hipikat, it can recommend colleagues who are topic experts that the user can contact for advice.

2.4 Mining artifact repositories

Modern software development teams store project artifacts in a variety of online repositories, such as a source revision control system and an issue reporting and tracking system. To provide control over changes to stored artifacts, these repositories usually maintain for each artifact a complete history of changes that it underwent. For long-running projects, these repositories provide a window into the project's past. Following Carl Sagan's saying "you have to know the past to understand the present," these repositories can serve as a rich source of information and past experiences useful in the present. In this section, I give an overview of previous approaches that mined artifact repositories to help developers in their ongoing tasks.

Mining source code repositories

In her seminal paper analyzing the use of configuration management tools as a coordination mechanism in software development [42], Grinter noted that the history of changes stored in the repository over time grew into an organizational memory of which artifacts changed as a result of a certain problem or enhancement. She reported that it was commonly used to learn what previous developers did:

> The memory gives [developers] the ability to leverage from the experiences of others. ... This mechanism of interaction can not be replaced by communication when the

[1] Letizia actually runs a breadth-first search until it is interrupted by the user following a link, when it stops and starts a new search from the new URL. Given typical browsing practices, this means the search usually only goes a few levels deep.

authors of the original software have left the organization. (p. 173)

A number of approaches attempt to make it easier to access the history stored in this memory. Atkins's *Version Editor* (VE) makes version history information available within an editing environment [6]. VE displays for the current line the revision in which the line was changed, together with its check-in description. The view of the file can be modified according to various visibility criteria. For example, VE can highlight lines changed in the current version, show lines deleted since a certain date, or lines touched by a particular set of changes.

CVSSearch by Chen et al. uses similar techniques but applies them in a different situation. It works as a search engine, where a developer can type in search terms and receive in response fragments of source code associated with CVS check-in comments in which the search terms occurred. Unlike VE, CVSSearch does not use just the comment of the latest check-in in which a line was changed: it tracks all changes as a file evolves, so that if a line has been modified, the comments associated with it for the purpose of searching are accumulated, although the most recent ones are given higher weight. CVSSearch can also give a developer a sense of how scattered an implementation of a feature is in the codebase by listing all files in which the search match occurred. It does not, however, try to group the files in a more meaningful way, such as those that were part of a same check-in.

Zimmerman et al. [125] and Ying et al. [124] applied data mining techniques to source repositories to find change patterns—sets of files that were changed together frequently in the past. These patterns can uncover program dependencies that would remain hidden using existing approaches, such as using structure encoded in the programming language (e.g., method calls or inheritance relationships). A tool can then make recommendations about potentially relevant files—based on the change patterns—to a developer working on a software change. For example, a developer checking in a set of changes into the source repository can be alerted if some members of common change patterns are missing from the check-in.

Bowman and Holt used source change data to determine a system's *ownership architecture*, which shows how developers are grouped into teams and the code that these teams have worked on [14]. The claimed benefits to the developers attempting to understand a system are identification of areas of expertise, non-functional dependencies, and quality estimates. There were also benefits to project managers: evaluating a system for risks of code abandonment and staffing problems, or overall developer coverage. However, this approach was demonstrated only by a proof-of-concept architecture built manually from data on the Linux project, and was not tested in practice.

Expertise Browser by Mockus and Herbsleb [78] also used source change data to identify developers and groups with experience in given elements of a software project. However, it determined expertise automatically from the version control system. The expertise database could be explored interactively, through an interface which would show for each file in the repository developers who had changed it in the past—or vice-versa, for each developer the files on which he or she had worked. The tool also offered visualization of the amount of modifications different developers performed on a given file, giving a graphical indication of relative expertise. Expertise Browser was deployed in a large, distributed software development project, where it was found to be the most useful to satellite sites new to the project or lacking sufficient breadth of expertise on location. It does not, however,

help a project newcomer, except by indicating to which colleagues should be most knowledgeable about a certain area of the code.

Expertise Recommender (ER) by McDonald and Ackerman [72] applied a similar approach to identifying expertise. However, ER worked as a recommender, where developers could make queries on certain project-specific keywords, and receive recommendation on people who have some expertise with a problem. It used an open architecture, with recommendation heuristics that could be customized to different organizational environments and practices. The prototype was tailored to a tech support organization and used as one of its main heuristics source change history (people who most recently checked in changes to a module).

Several approaches have used visualization of version history data as an aid for developers. SeeSoft [7] is a software visualization tool that can display a variety of graphical representations of source code. Among the representations using version history data are maps of files showing which programmers wrote which line, or areas of "fix-on-fixes": bug fixes which had to be fixed again because the original repair was faulty. While potentially a useful tool for project managers, these representations are arguably less useful for understanding the system itself.

Gulla also used versioning information to enhance traditional visualization techniques, for example, to show the amount of differences in two versions of the system, identify files that change often, or highlight those that changed together as part of a single check-in into the repository [44]. However, his focus was on software maintenance and configuration management tasks in projects that use multiple product configurations, not specifics of what the source code does.

Mining issue tracking databases

Compared to source repositories, issue tracking databases have only recently gained the attention of software engineering researchers, although so far mostly in the context of project management. For example, Sandusky et al. [102] have investigated networks of dependencies among problem reports in the Mozilla project. Their aim is to improve discovering problem dependencies as a problem management aid. In a related effort, Ripoche and Gasser [94] used an issue tracking database to automatically extract process models for open source projects.

The KDE open source project[2] problem reporting system uses text similarity measures drawn from information retrieval research to detect possible duplicate reports at submission time. Before a new report is entered into the database, the user is presented with a list of existing reports that have similar descriptions. The user can then check those reports to make sure his or her report is really on a new problem. While duplicate report detection is becoming a necessity for large open-source projects (the Mozilla project claims 30% of submitted problem reports are duplicates of existing ones, which adds a huge load on the maintainers), it can be difficult for the user submitting the report to recognize whether it is a duplicate if the symptoms are slightly different.

Čubranić and Murphy used issue-handling history to predict the developer that should be assigned to work on a particular problem report [28]. They applied machine learning techniques that were originally used for text categorization to build a model of each developer's expertise based

[2] `http://www.kde.org`

on the problem reports that he or she worked on in the past. This information consisted of textual description of each bug and the developer who solved it and was drawn from the issue tracking database. Once the "expertise" model was built, it could be applied to a new problem report to predict which developer should work on it.

Chapter 3

Hipikat

In this chapter, we present the details of our approach and its implementation in the Hipikat prototype, and we describe the practical application of this prototype to a large open-source project.

There are three different aspects of Hipikat, and we describe each in a separate section. First, we introduce the foundation of the Hipikat approach: the model of the project memory that we use and the mechanism for recommending artifacts from this memory (Section 3.1). Next, we describe the working Hipikat prototype that implements the project memory model and recommends artifacts in response to user queries (Section 3.2). Finally, we present the instantiation of the prototype for a concrete project, the Eclipse integrated development environment, and discuss work that would be necessary to apply it to a different target (Section 3.3).

The three aspects that we discuss in this chapter are largely independent. We could use the same tool in terms of the user interface, in particular how it presents recommendations and interacts with the user, but with a different project memory model. We could keep the same model, but use a different tool. And lastly, we could instantiate it for a project other than Eclipse.org.

3.1 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 the development history of a software project.

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 and that are a part of the project's history. 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 may be relevant to the task being performed.

These two functions can be 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.

3.1.1 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 [21]. 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[1]), source *file versions* (e.g., checked into a source repository such as CVS[2]), *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). (See Appendix A for a short review of open-source software development process, typical tools used, and artifacts created by open-source projects.) Figure 3.1 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.

Relationships (links) between the artifacts are established either from existing information about artifacts that is 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 meta-information 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 meta-information, such as when particular project artifacts were created or modified; for example, it is likely that the author of a bug fix checked in a source code revision(s) close to the time that the problem report was closed in the issue-tracking system. The specifics of our link inference algorithms are discussed in Section 3.2 below.

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 of 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

[1] `http://www.mozilla.org/projects/bugzilla`
[2] `http://www.cvshome.org`

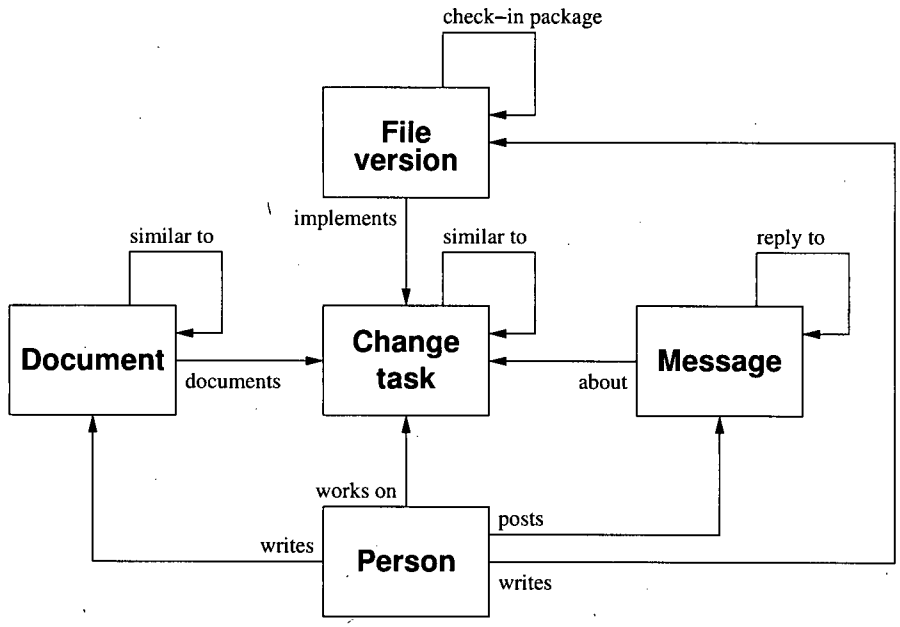

Figure 3.1: Artifact types in the Hipikat project memory and the relationships between them.

an existing issue. Other documentation may also 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.

3.1.2 Making recommendations

In the second component of Hipikat, selecting and presenting recommendations to a developer, the relationship links are used to select 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 and could coexist in a given implementation of our approach.

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 been completed previously within the same subsystem, or with a similar description. These artifacts are selected for recommendation by following `similar_to` links (see Figure 3.1) and are returned to the user to inspect.

Once the user 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 may have to be modified or understood for the task at hand. The completed similar tasks may also have related discussions about which design options were examined, and which decisions were made that might impact the task at hand.

3.2 The Hipikat tool

We have built a working Hipikat prototype that implements the project memory model and the functionality described in Section 3.1. The prototype has been designed to adapt easily to any open-source project that follows the development model described in Appendix A and that produces at least a subset of the artifact types contained in the project memory schema described above.

The Hipikat prototype is a client-server system. The client and the server communicate over a SOAP RPC protocol [15], with the recommendations returned by the server in an XML format, described in Section 3.2.1. The characteristics of the protocol allow language- and platform-independent implementation, and indeed high-quality freely-available libraries implementing the SOAP protocol and XML processing exist for a variety of languages and platforms. This allows a client to be written that is appropriate to a particular project and the development tools used by its members. For example, Mozilla developers mostly use standard Unix tools, such as Emacs, so the appropriate client for them should run within Emacs (written in Emacs Lisp), or perhaps as a plug-in within Mozilla itself (written in a combination of JavaScript and C++). On the other hand, Eclipse.org developers exclusively use the Eclipse IDE as their development environment, so for them the only reasonable option is to have a Hipikat client written in Java as an integral component of the Eclipse IDE.

In this section, we begin with the details of the client-server communication protocol (3.2.1). We then describe the current server implementation, including the link inference algorithms that we use (3.2.2). We conclude the section with a description of the implementation of the current Hipikat client, which is written as an Eclipse plug-in (3.2.3).

3.2.1 Hipikat client-server protocol

The client issues a request for recommendations to the server, and displays returned results to the user. The request is sent to invoke the method "getRecommendation" in the namespace "RecommendationFetcher" on the Hipikat server. There are three parameters in a "getRecommendation" request. Two of the parameters are required: the first identifies anonymously the user,[3] and the second identifies the artifact for which related items are sought. An optional third argument is intended to further describe the context of the query for additional tailoring of recommendations, although it is not used at this time. The server replies with a list of matches that the client then formats and presents in human-readable format.

The server replies with an XML-formatted list of matches which is modelled on the "What's

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

```
<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope
    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
    xmlns:xsd="http://www.w3.org/1999/XMLSchema">
<SOAP-ENV:Body>
  <ns1:getRecommendation xmlns:ns1="urn:RecommendationFetcher"
    SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
     <userId xsi:type="xsd:string">cb175</userId>
     <artifactKey xsi:type="xsd:string">bugzilla:20982</artifactKey>
     <contextKey xsi:type="xsd:string" xsi:null="true"/>
  </ns1:getRecommendation>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Figure 3.2: XML source of a sample request from Hipikat client.

Related" service provided by Alexa.com and available in all major web browsers. The recommendations are wrapped in a `RecommendationList` XML element (Figure 3.3).

Each item recommended by the server is represented as a `Recommendation` element, with children `Key`, `Name`, `Reason`, `Confidence`, `Created`, and `lastModified`. The `Key` element uniquely identifies the recommended artifact; it is used if the user decides to open it or make a subsequent query on it. The `Name` element is a human-readable description of the artifact. See Table 3.1 for a description of the name element construction for all artifact types.

Artifact type	**Key**	**Name**
CVS revision	`cvs`	file name : revision
Bug report	`bugzilla`	Bugzilla summary
Newsgroup article	`news`	subject (author)
Mail message	`mail`	subject author
Web page	`web`	Title (url)

Table 3.1: Construction of artifact keys and names for artifact types represented in Hipikat's project memory.

The `Reason` element describes why the item was recommended, and `Confidence` expresses the relative strength of this relationship. The confidence value can be descriptive, as in "High – checkin within five minutes" for a file version's link to a bug report, or numeric in the case of a text similarity measure. See Table 3.2 for a summary of confidence values for the corresponding reason element.

The `Created` element gives GMT-based date and time when the artifact was created. Element `lastModified` uses the same format for the last modification time of an artifact. In case the artifact was never modified (file revisions, news articles, and mail messages are immutable), the

```
<RecommendationList>
  <Recommendation>
    <key>cvs:dev.eclipse.org:/home/eclipse/
        org.eclipse.team.cvs.ui/src/org/eclipse/team/
        internal/ccvs/ui/actions/TagAction.java:1.13</key>
    <name>org.eclipse.team.cvs.ui/src/org/eclipse/team/internal/
        ccvs/ui/actions/TagAction.java:1.13</name>
    <Created&gt;2002-04-08 21:46:03&lt;/Created>
    <lastModified/>
    <reason>Bug ID in revision log</reason>
    <confidence>High</confidence>
  </Recommendation>
  <Recommendation>
    <key>bugzilla:12367</key>
    <name>[CVS Repo View] "Define Branch Tag" confusing?</name>
    <Created>2002-03-27 16:34:00</Created>
    <lastModified>2002-09-06 22:23:19</lastModified>
    <reason>Text similarity</reason>
    <confidence>0.6240631</confidence>
  </Recommendation>
  ...
</RecommendationList>
```

Figure 3.3: A sample response from Hipikat server.

`lastModified` element is empty.

3.2.2 Hipikat Server

The server has to implement three distinct functions:

1. **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. By the nature of the information sources that are used, some artifact types are immutable once created (for example, CVS revisions). Others can be modified, but never deleted (e.g., Bugzilla items). Finally, some artifact types are both modifiable and deletable (e.g., web pages).

2. **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).

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

Reason	Confidence
Bug ID in the check-in comment	High
Check-in close to bug resolution	High – within three minutes Medium high – within ten minutes Medium – within an hour Low – over an hour
Same check-in package	High – same check-in comment Medium high – checked-in within a minute of each other Medium – within three minutes of each other Medium low – over three minutes
Same email/newsgroup thread	N/A
Text similarity	Cosine similarity value (see *Text similarity matcher* in Section 3.2.2)

Table 3.2: Construction of confidence for artifact types.

As Figure 3.4 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 with each other, but instead share the access to the database, where the artifacts and artifact links are stored.

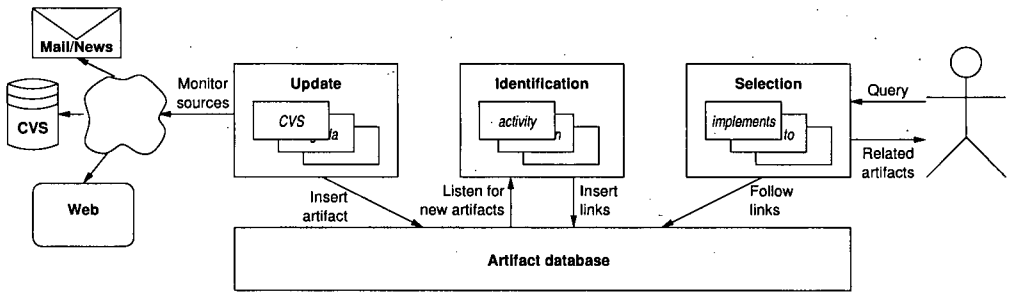

Figure 3.4: Hipikat server architecture

The server is written in Java and can be run either inside a web application engine, such as Tomcat,[4] or stand-alone. The project memory is stored in a MySQL relational database.[5]

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

[4] `http://jakarta.apache.org/tomcat`
[5] `http://www.mysql.org`

may be indexed, depending on the artifact's type. (See Table 3.3 for the full list of data Hipikat stores for each artifact type.) The indexed text is used for searching and making similarity comparisons, which we will describe below in the Identification section. Note that, specifically, contents of files in CVS repositories are not stored by Hipikat. There are two reasons for this. First, the storage requirements for keeping every version of every file would be huge without employing a space-saving algorithm such as that used by RCS. Second, and more importantly, for searching and making similarity comparisons on source code to be really useful, Hipikat would have to be aware of its syntax, that is, be able to parse the relevant programming language text. Even then, the exact treatment appropriate for different syntax elements (e.g., variables, methods, comments, etc.) and how similarity between different file revisions should be calculated is an open research question and outside the scope of this dissertation. Instead, for file revisions we only use their check-in comment for searching and similarity matching.

Artifact type	**Data stored in Hipikat**
File version (CVS)	Path Revision Author Date created Check-in comment
Change task (Bugzilla)	Bug id Reporter Summary Description *Other attributes (severity, OS, etc.)*
Web page	URL Title Text Last modification date
News article	Subject Author Text Date created Article id Follow up to ("References")
Email message	Subject Author Text Date created Message id In-response-to

Table 3.3: Artifact types and data stored about them.

Update

The update system has a separate module 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 [24] 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.

The actual implementation details of update modules are project dependent, and we leave their description until Section 3.3, when we describe the instantiation of the Hipikat prototype for the Eclipse.org project.

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 connection to the update system (Figure 3.5).

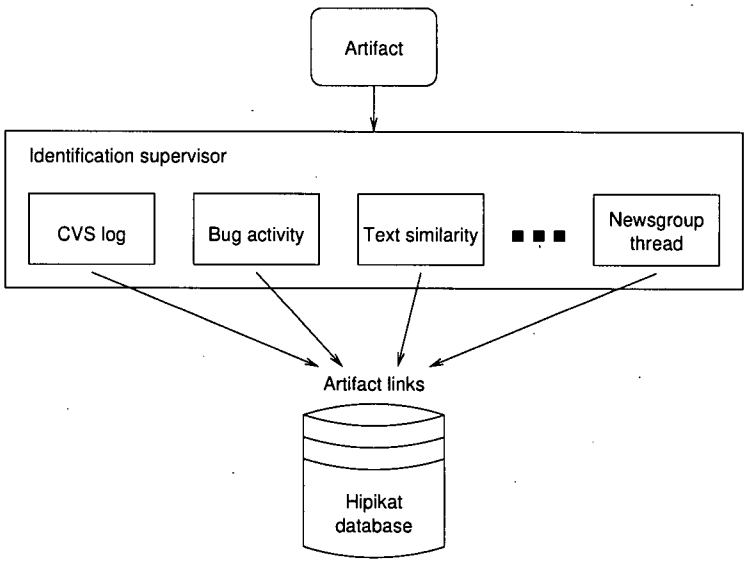

Figure 3.5: The identification subsystem

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: check-in comment matcher (*log-matcher*), check-in time 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 Figure 3.1. 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 post-processing, such as recalculating the text similarities. Table 3.4 lists existing matchers and artifact types for which each matcher is registered as a listener with the update system.

Matcher	Artifact type
Log matcher	`cvs`
Activity matcher	`bugzilla`
Text matcher	all artifact types
Check-in package matcher	`cvs`
Thread matcher	`news` and `mail`

Table 3.4: Identification modules and artifact types for which each matcher is registered as a listener with the update system

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. The exact regular expressions may vary with the project and the conventions that it uses, and we return to this issue when we describe the instantiation of Hipikat for the Eclipse.org project in Section 3.3. 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 comment check-in that are not enforced regularly. 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, similar to the strategy employed by Mockus, Fielding, and Herbsleb in their study of Mozilla development process [77]. Versions in each work unit are then 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 indexer The text indexer does not introduce any artifact links into the project memory, but instead performs "pre-processing" of artifacts necessary for text searching and the text similarity matcher.

In the indexing step, an artifact's text is transformed into a vector in the *term vector space*. In the vector space model, documents are represented as vectors of terms contained in a collection of documents, commonly called the *corpus*. In a corpus containing N terms, a document is represented by a vector in the N-dimensional space as follows:

$$Doc_j = (w_{1j}, w_{2j}, \ldots, w_{Nj}) \tag{3.1}$$

where w_{ij} is a value denoting the importance of term i in representing the concept of the document Doc_j.

The variants of the vector space model come primarily from the method of determining the value for each w_{ij}. We use the so-called *log-entropy* model [33], where the weight w_{ij} is calculated as a product of local weight L_{ij} of the term i in the document j, and global weight G_i of the term in the corpus.

The term's local weight factor reflects the intuitive reasoning that terms which appear more often in a document contribute more to its meaning. It is calculated as: $L_{ij} = log(1 + tf_{ij})$, where tf_{ij} is the term frequency, or the number of occurrences of term i in document j.

The term's global weight reflects the reasoning that terms which are limited to a few documents are more useful for discriminating documents from the rest of the corpus than terms that occur frequently across the entire corpus. In our model, it is defined as follows:

$$G_i = 1 - \frac{1}{log(N)} \sum_{j=1}^{N} p_{ij} log(p_{ij}) \tag{3.2}$$

where N is the number of documents in the collection, $p_{ij} = \frac{tf_{ij}}{d_i}$, and d_i is the number of documents containing term i.

In Hipikat, the text of new artifacts is indexed as they are added to the project memory. For each artifact—for example, a Bugzilla description and its comments, or a document on the project Web site—the constituent terms are extracted and their weight for the document calculated according to their frequency as described above. We follow the indexing practice accepted in the field of information retrieval and ignore certain common words (e.g., articles "a" and "the", prepositions such as "in", etc.) as defined in the SMART stopword list [54]. Furthermore, we perform *stemming* to map multiple grammar forms of a word (e.g., "work", "works", "working", "worked") to a single value. We use the standard Porter stemming algorithm [87].

Text similarity matcher The text similarity matcher determines the relationship between two documents by comparing their document vectors. We use the vector-space cosine measure common in information retrieval [101]:

$$sim(D_i, D_j) = \frac{D_i \cdot D_j}{\|D_i\| \|D_j\|} \tag{3.3}$$

Vectors D_i and D_j could come directly from Equation 3.1 for a document vector in the vector space model. However, this model has the drawback that the document vector representation is tied to the vocabulary: if two documents talk about the same thing but use different terms, they will not be similar under the vector space model. Instead, we use an extension of the vector space model called Latent Semantic Analysis (LSA) [31].

LSA essentially projects the matrix X of all documents in the corpus, formed by the document vectors Doc_j from Equation 3.1, to matrix $\hat{X}$—the *semantic space*. The idea behind this transformation of the document vector space is that because of synonymy (multiple words with the same meaning) and polysemy (multiple meanings of a single word) in natural languages, there is much noise in the matrix X, and that the projection to $\hat{X}$ reduces the noise. Therefore, even documents that use different terms to talk about same concepts can end up as similar vectors in the semantic space.

The text similarity matcher uses the vector-space cosine measure (Equation 3.3) to calculate similarity between pairs of documents, but document vectors come from the semantic space built by the LSA step. A list of neighbours to an artifact is built sorted by similarity, and `is_similar_to` links are created between the artifact and its most similar neighbours. New documents are projected into the database as they arrive and the similarity lists updated. For performance reasons, the list is normally truncated when similarity falls below a given threshold, or if it grows beyond a specified length (see Section 3.3.2).

"Classic" LSA is computationally costly for large document corpuses. Its essential step is singular value decomposition (SVD) of matrix X, whose time complexity for a sparse matrix of size $M \times N$ (M terms in the vocabulary and N documents) with c non-zero entries per column is $O(MNc)$.

To get around this limitation, a number of approaches have been proposed which use an approximation of SVD on X that is much faster to compute. A conceptually simple approach that is common in practice is to apply SVD to a sample S of documents from X, effectively reducing the N dimension of the matrix used in the SVD computation [34]. Once $\hat{S}$ has been calculated, it can be used to project all documents in X to $\hat{X}_S$ instead of $\hat{X}$. The criteria used to create the sample matrix S can have impact on the approximation accuracy. We use the method proposed by Jiang et al. and select documents from X with probabilities proportional to the square of their length [56].

Text searching The text similarity approach is also used in user-specified search queries: A user's query is treated just as another document vector, which is projected into the semantic space to sort the matching artifacts 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 comment that were checked-in within the same six minute window. The reason for the time window is to guard against developers (re)using the same, usually generic or empty, check-in comment for unrelated check-ins. We based the package matcher's heuristic and the length of time

window on Herbsleb et al.'s concept of a modification request [77], also used by German in his studies of open source project archives [41].

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 [51]) and reconstructs conversation threads of a newsgroup posting and subsequent replies. When an article is opened in a newsreader using just its article id (that is, when a Hipikat client opens it using the "news:" URL, rather than a user interactively from a subscribed newsgroup), the newsreader cannot navigate to preceding and subsequent articles. Instead, the user can receive the conversation thread in a Hipikat recommendation and open the individual articles this way.

A similar approach is applied to matching conversation threads in email archives. A standard email message contains a header referencing the id of the message to which it is a reply [27]. Alternately, 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.

Selection

The selection phase takes a set of candidate recommendations, and then 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 and selection heuristics, organized as a hierarchy of selection modules under the control of the selection supervisor (Figure 3.6). 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. (See Table 3.2 in Section 3.2.1 for a summary of confidence values for the corresponding reason.) If an artifact is reached by multiple links, the selection module will take that into account when giving a reason for the recommendation and 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 was assigned to the log id match because it is explicitly entered by the developer doing the check-in, so when it is present, it virtually always indicates the correct relationship between revision(s) and bug report(s).

The selection supervisor manages the selection modules and ensures that they are executed in the correct order. (See Table 3.5 for a summary of the module ordering.) Execution order is important because higher-level selection modules can rewrite the recommendation lists created by lower-level modules. For example, a user can mark a recommended artifact as particularly relevant to his or her

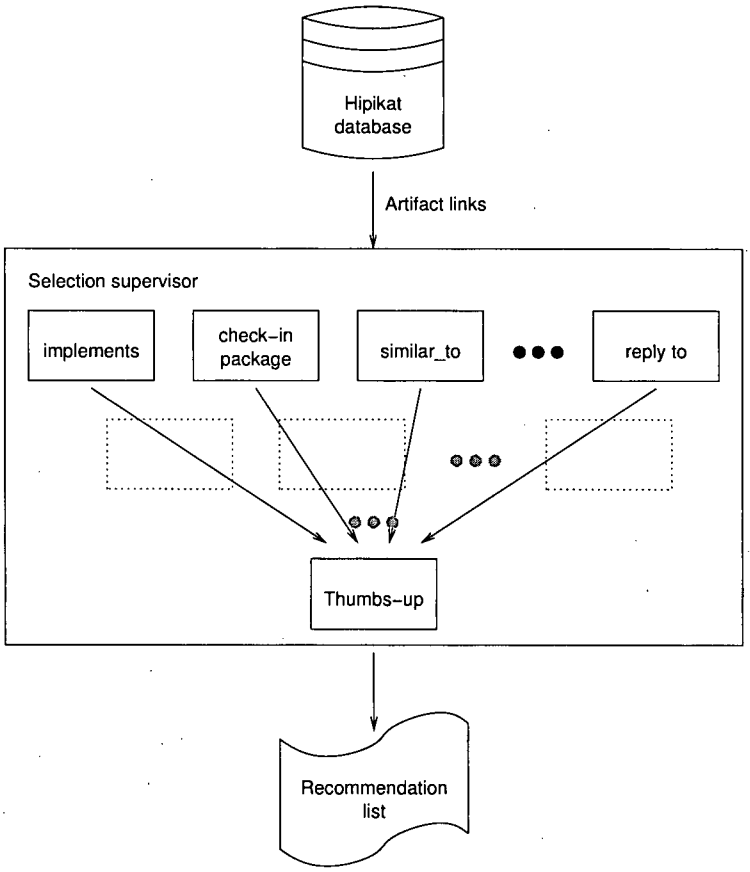

Figure 3.6: The selection subsystem

query ("give it the thumbs-up," see Section 3.2.3). This may eventually turn into a true collaborative filtering system (see Section 5.1.3 for further discussion), but for now we use it in a simpler way: artifacts in the recommendation list that were given thumbs-up by experts are moved to the top of the list. The criteria used to define the "expert" group vary according to the project, but in a typical open-source project, it would probably include the "core" developers.

Recommendations from all selection submodules to a given query are merged together before a final list is returned to the user. The initial order is a concatenation of recommendations returned by the top-level selector(s), but the client can reorder the list based on criteria such as the creation date of a recommended artifact.

Module	Description
Implements	Follows `implements/implemented-by` links between file revisions and change tasks
Check-in package	When a query in on a file revision, shows all other revisions that the check-in package matcher identified were checked-in together as part of the same package.
Reply-to	When a query is on a newsgroup article or a mail message, shows all other articles in the same conversation thread, as identified by the thread matcher.
Similar-to	Follows `similar-to` links between artifacts. Recommendations are ordered by collection, starting with change tasks, and continuing with file revisions, web documents, mail messages, and newsgroup articles.
Thumbs-up	Moves artifacts that were given "thumbs-up" by the user to the top of the recommendation list.

Table 3.5: Order in which selection modules are executed during the construction of a recommendation list.

Extending the server with new modules

Adding new heuristics and link types to the server is quite simple. The identification module that implements a new heuristic is written as a Java class that implements interface `ca.ubc.hipikat.server.identification.Identificator` and has to be registered with the identification supervisor. The registration is trivially done by adding the matcher's class to the server's configuration file and the will be dynamically loaded when the server is started.

If the identification heuristic creates a new link type, then a corresponding selection module has to be written which can interpret a link's attributes when providing the reason and confidence level for a recommendation based on the new link type.

Lastly, adding a new artifact type requires changes on both the server and client side. On the server side, if the new type comes from a new information source (for example, a text chat channel), then an update module would have to be written to monitor the new information source

and add artifacts of the new type to the artifact database. If the new type comes from an existing information source (e.g., a design specification posted on a web site in a special file format), then the corresponding update module (in this example, the web update) would have to be modified to handle the new artifact type when adding artifacts to the artifact database. In either case, identification modules would need to register with the update system as listeners for changes on artifacts of the new type. Finally, Hipikat client would need to be extended with the functionality for viewing artifacts of the new type. This could be as simple as invoking a user's web browser with an artifact's URL, if it has one, or as complicated as writing a specialized viewer.

3.2.3 Hipikat client(s)

As we mentioned earlier, a Hipikat client could be implemented in a variety of platforms and languages. An early prototype, used in an exploratory user study (described in Section 4.1) was written as a stand-alone Java application. However, based on the feedback from that study, we decided that the client should be as integrated as possible into the development environment used by developers in a given project. In this section, we present the current implementation, which is written as a plug-in for the Eclipse IDE. Eclipse is an easily extensible, open-source IDE that is increasingly popular with developers.

Eclipse's extensibility means that the Hipikat client appears seamlessly integrated into the IDE, including using it in combination with other software engineering tools plugged into Eclipse. For example, an Eclipse developer can access from the same Search dialog both Hipikat search and the Java search feature that is bundled with the default Eclipse distribution, simply by clicking on the appropriate tab as shown in Figure 3.7.

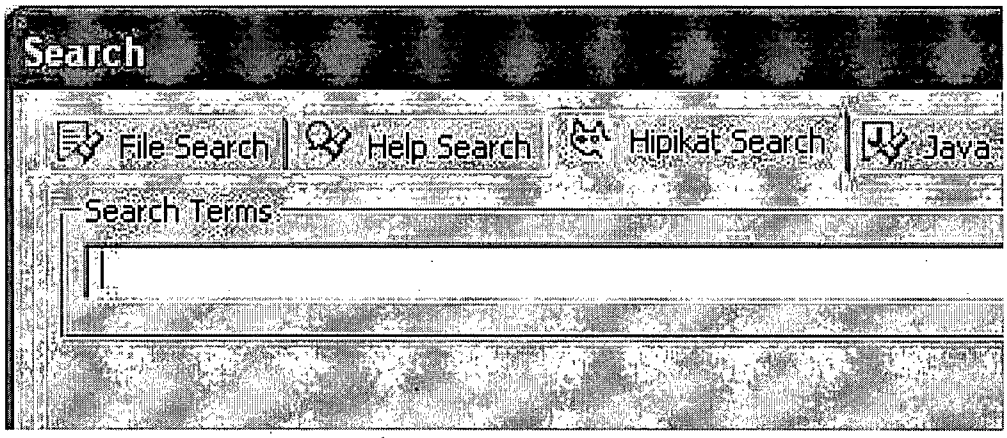

Figure 3.7: The Eclipse search dialog. The current tab is the Hipikat search. Other search types are available through neighbouring tabs. For example, the tab for Java syntax-aware search is immediately the right of Hipikat's.

User interaction

The basic user interface to Hipikat is simple: when a developer wants to find out more about an artifact in the Eclipse project workspace, he issues a query to the server and receives a list of related artifacts in response. These artifacts 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. (Possibilities for tracking user actions in the environment and gathering feedback on usefulness of recommendations are discussed in Section 5.1.3.)

When it is possible to make a query on an artifact in the IDE, there is an option "Query Hipikat" in the right-click context menu. For example, Hipikat knows about files in CVS. Therefore, the developer can right-click on any versioned file appearing in any representation in the IDE—in the directory tree in the Navigator view, as a class in Package Explorer, even as an entry in the revision list in the CVS Resource History view—and select "Query Hipikat" from the context menu (see Figure 3.8. 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.

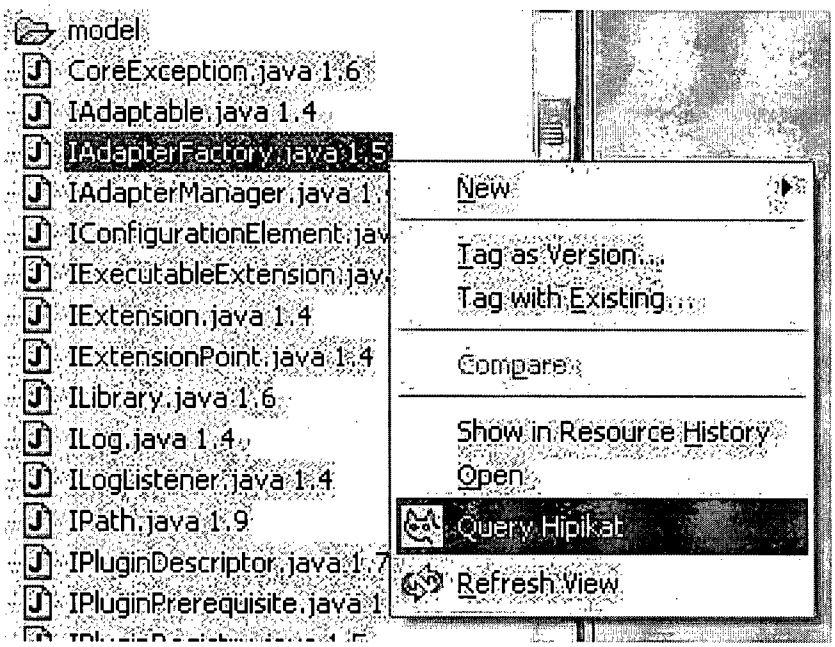

Figure 3.8: Option "Query Hipikat" in a file artifact context menu.

The identifier of the selected artifact is passed as the second argument in the request to the Hipikat server, described in Section 3.2.2. (See Table 3.6 for a full list of artifact types and places in the IDE where a Hipikat query can be made.)

The results of a query or search are displayed in a Hipikat *Results view* within Eclipse (see Figure 3.9). The view lists for each recommendation its type (Web page, news article, CVS revision,

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

Table 3.6: Places where Hipikat query can be done in Eclipse and artifact type that the query is on.

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 module (the "matcher") that reported a link. Because each recommendation includes the time when the recommended artifact was created, the user also has an option of reordering the list chronologically—in case he or she is interested in most recent artifact, for example.

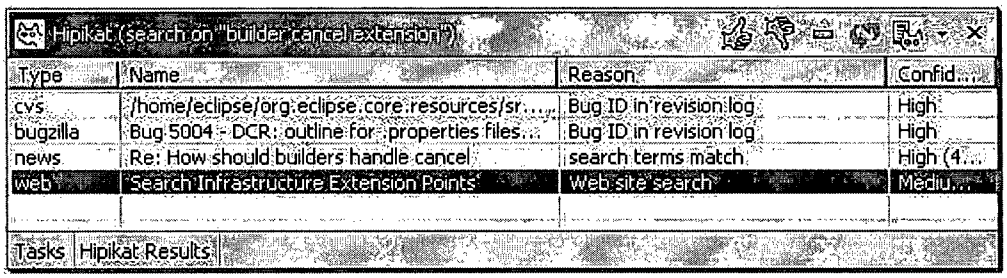

Figure 3.9: Hipikat results view.

Double-clicking on a recommendation in the results view opens the artifact for viewing. 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, shown in Figure 3.10. 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. Lastly, for a CVS file version, its differences to the preceding revision can be shown in the standard Eclipse "Compare with CVS revision" view. The intent of this feature is to make it easier for the developer to see the changes to the code that were made in the revision.

Previous Hipikat queries are accessible through the drop-down history menu at the right end of the view's toolbar (see Figure 3.11.

The toolbar also contains (going further from right to left) buttons to stop an ongoing query (query in progress is indicated by spinning arrows), and to indicate usefulness of a recommendation

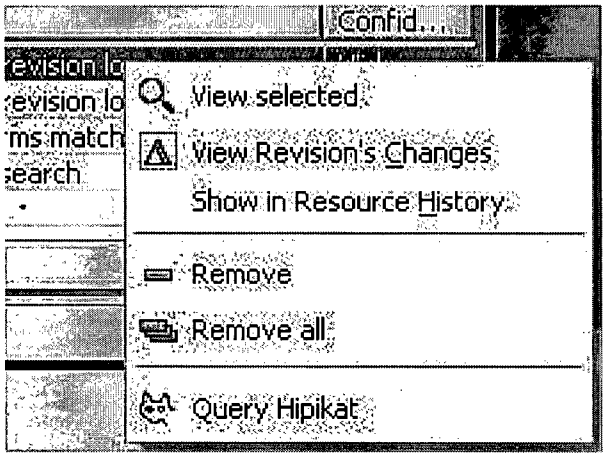

Figure 3.10: The context menu in Hipikat Results view.

Figure 3.11: The search history menu in Hipikat results view.

(or lack thereof) by giving it a thumbs up or down. Currently, the only visible effect of ranking an item is moving it to the top or the bottom of the recommendation list view, although we plan to eventually use it to refine the recommendations by a collaborative filtering step, an issue further discussed in Section 5.1.3.

3.3 Hipikat instantiation for Eclipse.org

We have instantiated the Hipikat prototype described in the preceding section to a large open-source project, Eclipse.org.[6] We have so far discussed Hipikat's implementation at a fairly general level of algorithms and heuristics. In this section, we present the details of adaptations that were necessary to fit Hipikat into the Eclipse.org project. We believe that many of these techniques could be adapted with little changes to other open-source software projects, since the information sources we used are quite common (e.g., Bugzilla is the de-facto standard issue-tracking system in the world of open source software). In many cases only the server addresses and URL paths would be different, which is only a matter of editing a configuration file.

We first describe how the update phase was customized to access the information sources used by the project, followed by the customization of the identification heuristics. We then continue

[6]We use the term "Eclipse.org" to distinguish the project from the product, the Eclipse integrated development environment.

with the description of database size and the system's performance. We conclude the section with a description of an Eclipse plug-in that we wrote to access and query Bugzilla servers directly from the IDE, instead of the standard web-based front end. This plug-in is used by the Hipikat client to view Bugzilla reports and was used in the study presented in Section 4.2.

3.3.1 Artifact update

The Eclipse project uses the following artifact sources: Bugzilla for issue tracking, CVS for source repository, email lists and Usenet newsgroups for developer discussion, and a web site for the documentation (user and developer guides, code-writing tutorials, development plans, etc.).

Bugzilla

The issue-tracking system uses a standard distribution of Bugzilla (with minor differences in visual layout).[7] A Bugzilla server runs as a web application; clients communicate with it by invoking predefined URLs on the server machine. These requests span in functionality from showing a user interface (e.g., a form for filling in bug search parameters) to displaying results of an operation (e.g., retrieving a list of bugs that match a search criteria).

We wrote a Bugzilla update module that monitors the activity in a Bugzilla database by connecting to a Bugzilla server like any client and processing the server output to bring the Hipikat artifact database in step with recent changes in Bugzilla's. The update module starts its periodic update by sending a request to the Bugzilla server that asks for a listing of bugs that have changed in the last n days, where n is calculated since the last update.[8] The server's response lists all bugs that match the request's criterion. The list is in HTML format and includes such bug information as a one-line summary of the bug and its severity and status, as shown in Figure 3.12.[9] The id's of bugs in the list can then be extracted by matching for the text "<a href="show_bug.cgi?id=*bugid*">" in the list's HTML source. Those bugs are then retrieved,[10] parsed, and added to the project memory.

As of September 17, 2004, the Hipikat project memory for Eclipse contained 69,375 bug reports, with 260,894 additional comments posted on the reports. An update catching up the project memory with the Bugzilla repository takes approximately 35 minutes when done daily (an average of 71 new bugs and 247 comments per day in the first seventeen days of September 2004).

[7] The Eclipse Bugzilla server is located at the URL `https://bugs.eclipse.org/bugs`.

[8] The number of days n is encoded in the URL of the request: `https://bugs.eclipse.org/bugs/buglist.cgi?changedin=`n`&cmdtype=doit`

[9] Recent versions of Bugzilla server software include functionality to return this list in a standard XML format that is easier to work with. The Eclipse project uses an older version of the software, so parsing the HTML output was our only option.

[10] The bug reports are retrieved using URLs of the form `https://bugs.eclipse.org/bugs/show_bug.cgi?id=`*bugid*

```
<tr class="bz_enhancement bz_P3 ">

  <td>
    <a href="show_bug.cgi?id=39424">39424</a>                    ← bug id
  </td>

  <td><nobr>enh</nobr>
  </td>
  <td><nobr>P3</nobr>
  </td>
  <td><nobr>All</nobr>
  </td>
  <td><nobr>jdt-ui-inbox@eclipse.org</nobr>
  </td>
  <td><nobr>RESO</nobr>
  </td>
  <td><nobr>DUPL</nobr>
  </td>
  <td>Add abstract method quickfix
  </td>

</tr>

<tr class="bz_normal bz_P3 ">

  <td>
    <a href="show_bug.cgi?id=39473">39473</a>
  </td>
...
```

Figure 3.12: Source of a bug listing.

CVS source repository

Eclipse uses CVS for its source code repository and version management [12]. The server allows anonymous access using the "pserver" CVS remote access protocol [30].[11] Remote users have a *local copy* of the repository in their workspace, which contains a particular configuration of file versions (one version of each file—typically, the most recent one). Changes to files in a local copy can be *committed* back to the repository using a CVS client. Committed changes are disseminated to other existing local copies when their owners *update* them to bring them in sync with the repository.

We wrote a CVS update module that works works by mimicking the interaction with a CVS server. The update module keeps its own local copy of the entire project and periodically updates the copy using a standard CVS command-line client. Under normal usage this may involve resolving

[11]The project's CVS server is located on host `dev.eclipse.org`, with the repository root at the directory `/home/eclipse`.

conflicts if a file has been changed both locally and in the repository. However, because our local copy is not modified, no conflicts can occur. All changes happen only on the server and the local copy is simply brought up to date with it. This may include replacing an existing file with a newer version, checking out a newly-created file that does not exist in the local copy, or deleting a file from the local copy that has been removed from the repository.

Hipikat monitors the diagnostic output of the command-line client to see which files were modified, deleted, or created. For each of those files, Hipikat then requests a history (*log*) from the CVS server. The log lists all versions of a given file, with details such as the author, creation date, and check-in comment (see Figure 3.13). This list is parsed to extract the data for each version and add the corresponding new artifacts to the project memory.

```
RCS file: /home/eclipse/org.eclipse.core.boot/plugin.xml,v
Working file: plugin.xml
head: 1.28
branch:
locks: strict
access list:
symbolic names:
  ...
----------------------------
revision 1.28
date: 2004/01/15 15:47:07;  author: prapicau;  state: Exp;  lines: +1 -4
Delete all content from org.eclipse.core.boot
----------------------------
revision 1.27
date: 2003/11/25 21:29:22;  author: dj;  state: Exp;  lines: +0 -1
Merge checkpoint.
----------------------------
revision 1.26
date: 2003/11/06 14:44:33;  author: dj;  state: Exp;  lines: +1 -0
Added "eclipse" precondition tag to the plugin manifiest.
```

Figure 3.13: Portion of log

We adopted an off-the-shelf approach when we decided to re-use the standard Unix CVS command line client instead of implementing our own solution to monitor changes in CVS repositories.[12] This allowed us to save the time to get the CVS repository update working. The CVS client-server protocol is sufficiently complicated to convince us against reimplementing the standard command line client, which is proven in practice and maintained by the CVS development team. One drawback is that the Hipikat server has to keep a full local copy of the Eclipse CVS repository—which takes up 960MB of disk space—even though the actual contents of files are not

[12]The client is included as a standard part of most Linux distributions, and is also available for download at `http://www.cvshome.org`.

needed. A solution that would save on disk space while allowing us to continue using the command line client would be for the Hipikat server to truncate all files in the local copy to zero length, but preserve the timestamp, which is all the command line client uses during the CVS update to determine what has changed locally.

As of September 17, 2004, there were 365,004 revisions of 47,895 files in 297 projects stored in the project memory. An update catching up the group memory with the CVS repository takes approximately 90 minutes when performed daily (an average of 232 new revisions per day in the first seventeen days of September 2004). A lot of this time is taken up retrieving logs of files that were changed. Running just the update from the command line takes up about 30 minutes.

Usenet newsgroups

Eclipse uses Usenet newsgroups as a public forum where users and third-party developers can ask questions from the Eclipse team. The newsgroups are accessed using the standard NNTP protocol (RFC 977).[13]

Newsgroup update is rather straightforward, since the NNTP protocol provides a request to the server to list id's of all articles in a given set of newsgroups that have been posted since a particular date. We wrote a newsgroup update module that periodically connects to the Eclipse news server and issues such a request for articles newer than the last update. The update module then retrieves from the server each article in this list, parses the meta-data and the article body, and enters the corresponding artifact into the database.

There are three Eclipse IDE-related newsgroups hosted by Eclipse.org: *eclipse.platform* (targeted at third-party plug-in builders), *eclipse.platform.swt*, and *eclipse.tools.jdt* (for discussion of the user interface toolkit and Java development tools, respectively).

Email lists

Eclipse uses mailing lists for communication by developers actually working or otherwise contributing to day-to-day development. There is a wide range of lists, divided by topic, such as "jdt-debug-dev" for discussion of Eclipse's Java debugging component.

These mailing lists are automatically archived using a program called Mailman.[14] Archives are accessible through a web front-end. Message id's within each list start from "00000" and go up sequentially in the order emails are received at the list's mail server. The mail update module that we wrote works along the lines of the Bugzilla update module described earlier. It periodically connects to the URL for the index of messages posted to each project mailing list,[15] using the same HTTP protocol as any web browser. Messages in the index are listed in reverse chronological order, so the update module only needs to extract the current highest message number, to know which

[13]The Eclipse newsgroups are stored on the Eclipse.org news server, `news.eclipse.org`, and are access-controlled by issuing a username and a password to registered users.

[14]`www.gnu.org/software/mailman/mailman.html`

[15]List index is available at URL in the form `http://dev.eclipse.org/mhonarc/lists/-list-name/maillist.html`.

messages have been added since the last update. It then accesses the URL for each new message [16] and parses the HTML page with the message to extract the message contents.

Interestingly enough, the message's page contains—embedded as comments—most of the original messages headers that are not visible in the page, such as the message's id and the id of the message to which it is a response, if any. Knowing each message's id is useful because that way the threading heuristic used for newsgroup articles can be applied to email messages as well. Furthermore, although the email address for the author of the message is anonymized in the visible page, the original "From:" header is also captured in the HTML source (trivially scrambled using the Rot13 cypher), and the author can be correctly identified that way.

Web site

The project's web site [17] includes project news, development plans, tutorial articles, FAQs, and online documentation. We wrote a web update module that monitors the changes to the project web site by operating as a simple web spider [24]. The module periodically connects to the project web server, starting at the main page, and follows links to text and HTML documents with URL's in the Eclipse.org domain. Some URL paths are "blacklisted"; those links are not followed because they lead to CGI scripts such as the mailing list archive or the bug repository, whose artifacts are already processed by other means, or to irrelevant sections of the web site, such as the downloads. Figure 3.14 contains the current URL blacklist.

Path	Reason
/viewcvs	Web interface to the CVS repository *Note: Regular expression* `/viewcvs/index.cgi/~checkout~/.*\.htm` *is used to identify URLs that are* exceptions *to the blacklist. Links matching the exception pattern are followed during the update because it is used to refer to the newest revision of various plugin-related development documentation, such as design requirements or development plans.*
/mhonarc	Web interface to the mailing lists archives
/mailman	Web interface to administering the mailing lists
/newsportal	Web interface to the newsgroup archives
/documentation/html	Online documentation
/bugs	Web interface to Bugzilla
/downloads/	Links to Eclipse binaries for download

Figure 3.14: URL paths that are ignored by the crawler.

[16] Individual emails are accessed at a URL of the form `http://dev.eclipse.org/mhonarc/lists/`*`list-name`*`/msg5-digit-id.html`

[17] `www.eclipse.org`

3.3.2 Identification heuristics

In Section 3.2.2, we described the identification heuristics ("matchers") used in the current Hipikat prototype. In this section, we give the specific parameters of those matchers that needed to be adapted to the work practices and conventions used in the Eclipse project. We believe that applying Hipikat to another large project using open source-style development methodology and a subset of the information sources used in the Eclipse project would involve only minimally changing these parameters.

Log-matcher As mentioned in Section 3.2.2, the exact regular expressions used by the log-matcher module may vary from project to project, depending on conventions adopted by its member developers. Based on our observations of check-in comments in the Eclipse project and informal conversations with developers on the project team, we determined the following typical cases ("bug-ids" is a number, defined as a sequence of digits, or a sequence of numbers separated by commas, spaces, "and," or "&"):[18]

- the comment consists only of *bug-ids*: regular expression "`\[`*bug-ids*`\]`"
- phrase "*#bug-ids*": "`#\s*`*bug-ids*"
- phrases "bug" or "bugs" followed by *bug-ids*: "`[bB]ugs?␣+`*bug-ids*"
- phrases "fix," "fix for," "fixed," "fixes" or "fixes for" followed by *bug-ids*: "`[Ff]ix(?:e[ds])?␣+(?:for)?[^A-Za-z0-9]*`*bug-ids*"
- *bug-ids* at the very beginning of the comment: "`^`*bug-ids*`[:]␣`"

Activity-matcher The customizations to the activity-matcher heuristic deal with the length of time window in which to look for a CVS check-in corresponding to the Bugzilla activity. We start by looking at all check-ins by a given developer within a six hour window ending ten minutes *after* the bug activity. The reason for extending the window a little following the activity is that we have observed some developers close the report in the bug-tracking system just prior to committing the fix to the repository. We use a six-hour window because we observed some developers (or often, interestingly, developer sub-teams) close the bugs en-masse at some point in their day. The length of the window is a compromise to catch this kind of bug management without yielding such a large number of matches that they would be hard to evaluate in a recommendation list. All versions checked-in within the six-hour window are sorted by time difference between their check-in and the bug's resolution; the nearest version and all the eligible versions checked in within the next three minutes are identified for linking (of type `implements`) to the bug report (see Figure 3.15). The identified links are further filtered by the rule that the activity matcher will link a file version to multiple bug reports only if the activities that triggered the initial match occurred within two minutes of each other, otherwise the one later in time is not linked.

[18] That is, *bug-ids* stands for the following regular expression: `(\d+((?:(?:,␣*)|(?:,?␣*and␣*)|(?:␣+)|(?:␣*&␣*))\d+)*\b(?!\.\d+)`

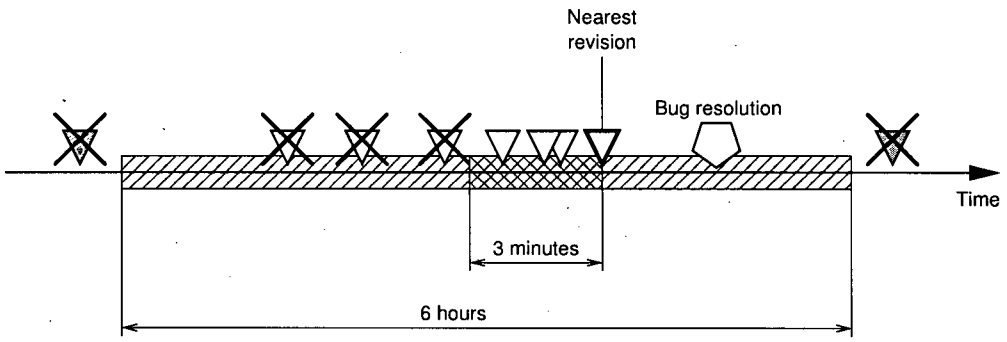

Figure 3.15: Schematic of the activity matcher's selecting of file versions to be linked to a bug report. The hatched pattern shows the six-hour window surrounding the bug report's resolution. Only those file versions that were checked-in within that window are considered for linking (white triangles). The version that is the nearest in time to the bug's closing defines a three-minute window for a "check-in package" (shown in cross-hatched pattern). Versions checked-in within that window are actually linked by the activity matcher to the bug.

3.3.3 Project memory database

Hipikat project memory is stored in an MySQL[19] database. The database consists of 58 SQL tables and takes up 800MB of disk space.

The host running the database is a 500MHz Pentium PC with 500MB of RAM. It runs Fedora 2 distribution of GNU/Linux (kernel version 2.6.6). The version of the MySQL database that is installed on the host is 3.23.58. The hardware and the software are essentially that of a stock PC workstation; no optimization or tailoring to database server applications have been conducted.

The mean time to create a recommendation list for all queries performed by the participants in the Eclipse study (Section 4.2) is 7.7 seconds with standard deviation of 8.2 seconds. No investigations of the causes for the relatively high standard deviation have been conducted, although it is likely that there are two major causes: network delays and the server that was not optimized for database applications.

3.3.4 Bugzilla front-end as an Eclipse plug-in

We wrote a plug-in that makes it possible to interact with Bugzilla from within the IDE, rather than through a separate web browser. To a plug-in user, Bugzilla search is similar to other searches in the IDE (see Figure 3.17 for a screenshot of the search dialog and Figure 3.16 for a view listing the results of the search).

A bug can be opened in an editor in the IDE (see Figure 3.18). The user can then edit the bug with functionality similar to the web-based client, for example post new comments or change various attributes of the report.[20] Most importantly, and the reason we wrote the plug-in in the first

[19] `http://www.mysql.com`

[20] Some functionality is missing at this time, in particular adding users to the CC list. We did not focus on

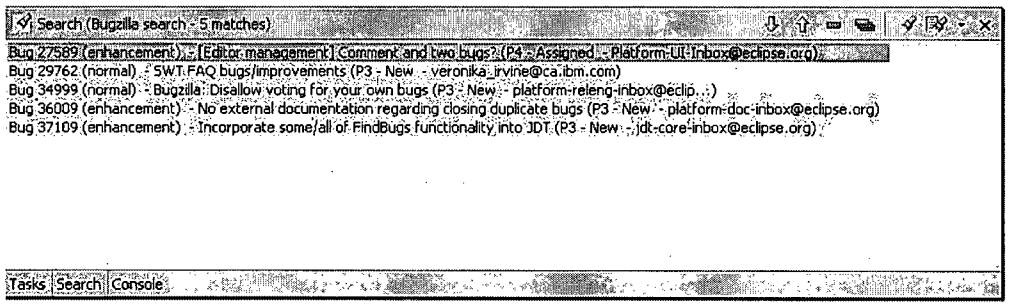

Figure 3.16: Bugzilla search results.

place, a user can issue Hipikat queries from the editor. (The integration is two-way: a double-click on a Hipikat recommendation of a bug report will open the report in the editor.)

Access to Bugzilla from within the Eclipse IDE started off as part of the Hipikat plug-in, but as it became a more full-featured Bugzilla front-end, we have spun it off as a separate Eclipse plug-in. As of September 17, 2004, it has been downloaded 382 times.

3.4 Summary

In this chapter, we have described the details of our approach and its implementation in the Hipikat prototype. We described three different aspects of Hipikat: the model of the project memory that we use and the mechanism for recommending artifact from this memory; the working Hipikat prototype that implements the project memory model and recommends artifacts in response to user queries; and the instantiation of the prototype for a concrete project, the Eclipse integrated development environment.

The project memory model consists of the project artifacts themselves and of links between those artifacts indicating relationships. Relationships between the artifacts are established either from existing information about artifacts that is available from the project management tools or that is inferred by Hipikat.The relationship links are used to select relevant artifacts in response to a query.

The Hipikat prototype implements the project memory model and a number of heuristics that identify links between related artifacts and use those links to make recommendations. The prototype has been designed to adapt easily to any open-source project that follows the typical open-source development model (see Appendix A) and that produces at least a subset of the artifact types contained in the project memory model. We present the description of adaptations that were necessary to fit the prototype to a large open-source project. We include the performance of the system and storage space requirements for over two years of project memory.

---

it because this feature was not needed in our user studies.

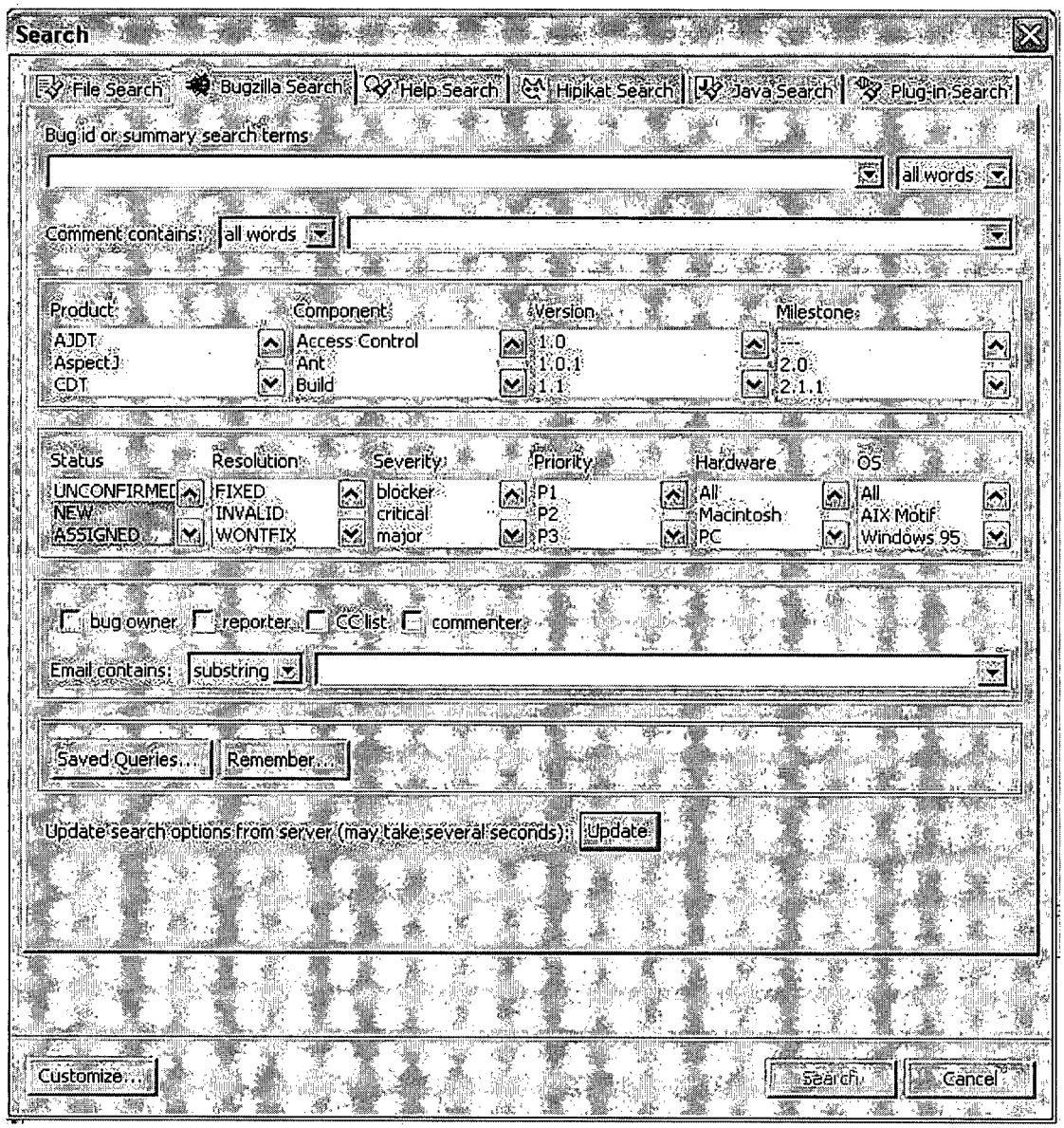

Figure 3.17: Bugzilla search dialog.

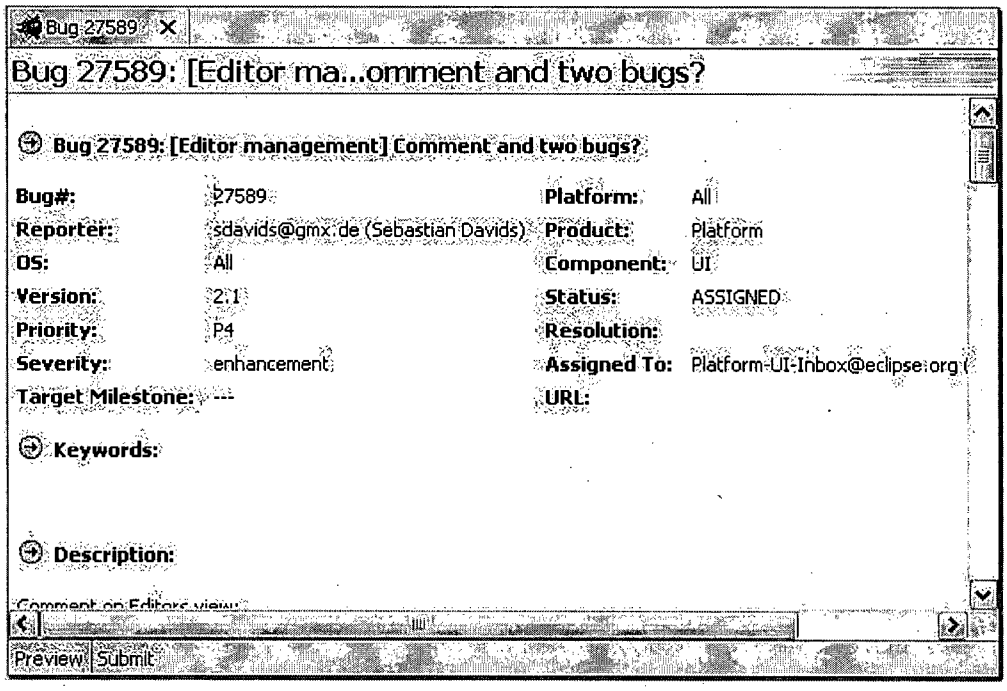

Figure 3.18: Viewing a bug.

Chapter 4

Validation

The hypotheses of this research are premised on the idea that the collection of all artifacts created in the course of development of a software system implicitly forms a group memory. We make three claims:

1. that newcomer software developers can use information from the project memory about past modifications completed on the project to help them effectively perform modification tasks to the system;

2. that this project memory can be built largely automatically, requiring minimal adjustments in work practices of software developers;

3. that the automatically-built project memory can recommend useful artifacts, in particular past modifications, that identify target classes, reusable code, or design decisions and other information pertinent to the current change task.

The first claim is supported through two case studies. The first study was more exploratory in nature and used a project memory that was built by hand from a project's authentic online repositories (Section 4.1). The results of the first study drove Hipikat's development and led to the second study: a multiple case design in which Hipikat was used to add features to a large open-source software system (Section 4.2).

The second claim is supported through the implementation of Hipikat used in the second study. This implementation, already described in Chapter 3, builds the project memory by mining the kinds of archives that typically exist in open-source projects and requires no adjustments to the developers' working practices.

We tested the third claim by manually evaluating the quality of Hipikat's suggestions for twenty change tasks selected at random from the set of modifications implemented in a major release of a large open-source software project. The details of this evaluation are presented in the final section of this chapter.

4.1 The Avid study

As the first step in evaluating the first two claims introduced at the start of this chapter, we conducted an exploratory study in which participants used an early Hipikat prototype to perform modifications to an unfamiliar software system. We focused on whether recommendations of past modifications were of any help to developers working on a change task, and if so, which kinds of recommendations were used. We also wanted to determine if the way we presented the recommendations was adequate and if there were other user interface issues that we needed to take into account.

Since the focus of the study was on the *usefulness* of Hipikat's recommendations, we decided to build the project memory by hand. This way we could ensure that the project memory contained recommendations that were relevant to the change tasks, and that were representative of various recommendation heuristics that we were implementing for the automatic version. We used existing artifacts from the development of the target system, but determined the links manually (see Section 4.1.1 for details). The Hipikat server therefore consisted only of the selection stage (see Section 3.1.2), while we took on the role of update and identification stages. This approach allowed us to focus on getting a working client front-end and to test the user interface and recommendation heuristics with users before committing to implementing them on the server end.

In this section, we describe the design of the study, including the details of how the project memory was created and the user interface that was used to access it. We then describe how we selected the study participants, and the procedures we used to run the study, followed by the study results and their implications for the next step of this research.

4.1.1 Design

Since we were in an early stage of our research, it was not possible to define and isolate a set of variables to be used in an experiment. Instead, we chose a case study format in which we investigated the use of Hipikat by software engineers who were implementing changes to a software system with which they were not previously familiar. Although we did not conduct our study in a real workplace context, to ensure the setting was as realistic as possible, we decided to use an existing software system as the study's target.

To maximize the potential for data collection, we conducted the study as part of an assignment in a graduate software engineering class. The assignment asked students to implement the changes using two assigned program understanding tools, one of which was the Hipikat prototype, and to write a report describing their experience. The students worked in pairs, in order to facilitate feedback, debate, and idea exchange while working on unfamiliar software system using new software tools [82].

Participation in the study was voluntary: students who so chose could share their assignment reports with us and take part in a follow-up interview. We relied on the participants' own descriptions of their experiences with Hipikat as an indication of its usefulness and as a generator of ideas for improvements to the prototype. We decided not to analyze the code in their implementation of the changes because our focus was not on their performance with or without Hipikat, but rather on instances where the tool helped (or failed to help) in their task.

Furthermore, since the students could choose whether to participate in the study *after* they had completed the assignment, we might not have observed them while they were working on the change tasks, and thus could not use that as an additional source of data. We feel that these trade-offs were appropriate to the kinds of questions we were asking at this stage of the research, and for the amount of resources required to set up and run the study, and collect and analyze the data.

The target system: Avid visualizer

Having to manually create the memory meant that we had to balance two requirements about the size of the target system that we used in the study. On the one hand, it had to be complex enough that modifying it was not trivial. On the other hand, we needed to have a fairly complete coverage of the code by Hipikat's recommendations: we felt that if participants encountered significant areas of the target system that had no recommendations from Hipikat, they would simply give up on the tool. However, a target system that was too large would require a huge effort to create the appropriate coverage in the project memory, when most of it would probably go unseen in the study. We therefore settled on a relatively small, but non-trivial system that was under development in our lab at the time, and for which we had the development history and easy access to developers for consultation.

The software system that the participants worked on in the study was the Avid visualization tool. Avid is a tool for visualizing the operation of Java systems at the architectural level [117]. It is an off-line visualizer which uses data that has been previously collected during a software system's execution (such as object allocation and deallocation, method calls, etc.) and applies concepts from the field of computer animation to display that information to a user. Avid uses a sequence of "cels," corresponding to individual animation frames, to represent the information collected across a system's execution. Each cel displays abstracted dynamic information representing both a particular point in the system's execution and the history of the execution to that point. Showing cels in quick succession creates an animation of the system's execution. Using Avid, a software engineer can navigate both forwards and backwards through the cels comprising views on the execution, and gain an understanding of the system's dynamic behaviour.

In addition, the information is visualized in terms of a high-level view of the system selected by the user as useful for the task being performed. This view is very flexible: an entity in it (called a *category* in Avid's terminology) can consist of a single class of interest, a small group of related classes corresponding to a functional subsystem, or a large set of classes that are less important to the current task. Figure 4.1 shows a screenshot of an Avid visualization. Several high-level entities can be seen, with arrows between them representing method interactions and object creation.

Avid was developed at the Software Practices lab (SPL) at the University of British Columbia. Avid is written in Java and has 12,853 non-comment, non-blank lines of code organized in 177 classes and 16 packages.

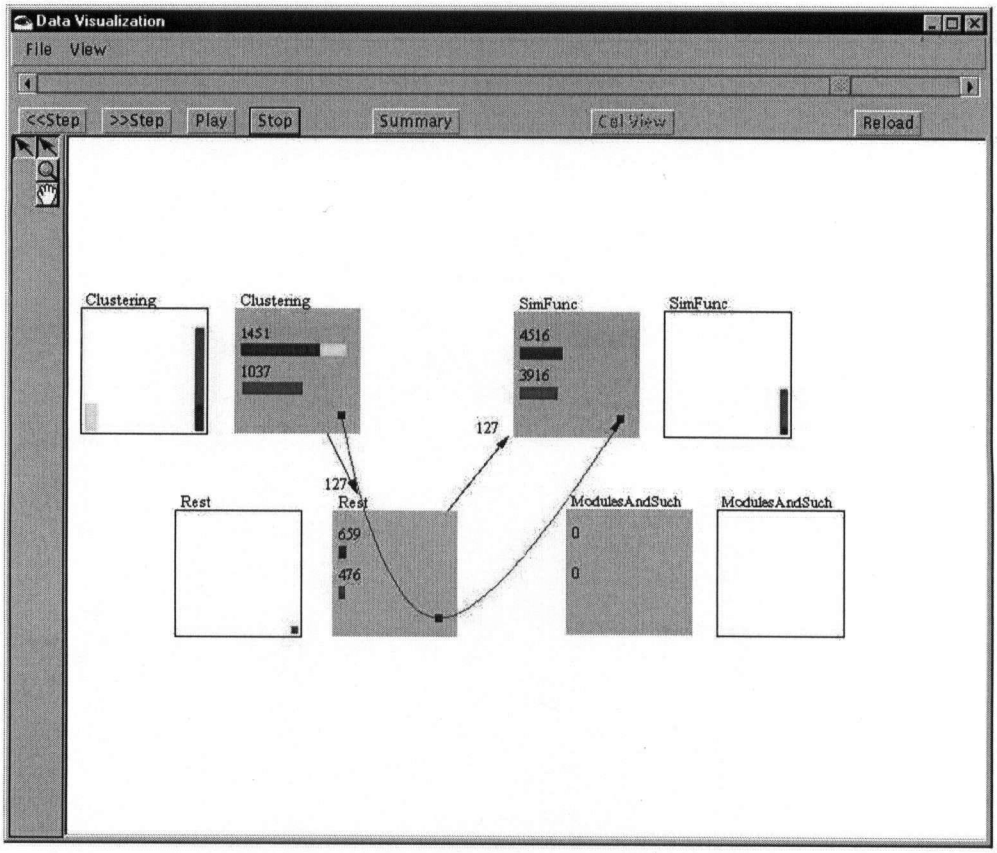

Figure 4.1: The Avid visualizer.

Hipikat project memory

As described in Section 3.1, the Hipikat project memory consists of project development artifacts and links between them. For this study, we had access to Avid's CVS repository, documentation, and project-related emails, and we inserted those artifacts into the project memory. Since the Avid project did not use a bug-tracking system, we analyzed its CVS repository, extracted about two dozen distinct change tasks from its history—including both bug fixes and new functionality—entered these tasks as items in a Bugzilla database, and added the items to the project memory.

We then manually formed the links between the artifacts in the group memory. We followed the principles described in Section 3.1: revisions were linked to change tasks based on text similarity of the check-in comments to task descriptions, documentation was associated through text similarity, and associations between change tasks were inferred using text similarity and overlap in source files that were changed as part of a task. These links were realistic: they contained both relevant and irrelevant suggestions that were based on the information recorded in the CVS repository. For instance, change tasks that contained similar words in their description were sometimes about something totally different, and groups of CVS check-ins often included some revisions whose modification was

not related to the check-in comment or to the rest of the group.

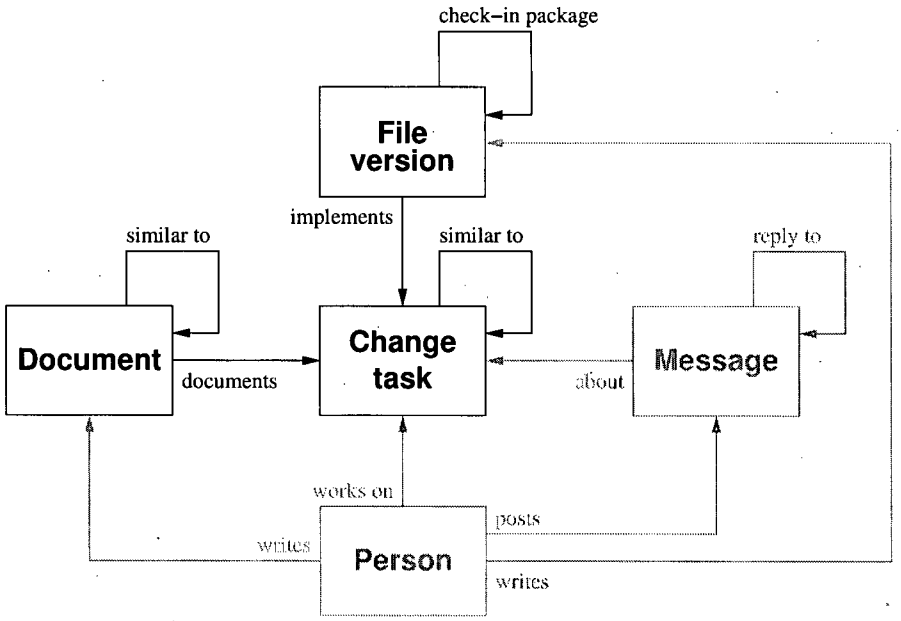

Figure 4.2: Hipikat project memory schema used in the exploratory study.

Study tasks

We selected as study tasks two previously completed changes from the development branch of the project:

Task A: Excluding object age information from the summary file. To visualize the execution of Java programs, Avid requires, amongst other inputs, a file containing summarized information about the events generated during the execution of a Java program. The summary file is built from the event trace file created during the program's execution and contains information such as the number of calls and the number of objects allocated or deallocated up to a certain point in the trace file.

The summary file also contains information about the age of objects at deallocation time. This information is voluminous, and experience with the visualizer has showed that it is not always used by the developers. The task required the students to add a command-line option to Avid to exclude object age from the summary file, and accordingly to not display object age histograms in the visualization when using such summary files as input.

Task B: Reporting detailed call information from call summary arrows. In the visualizer, each black directed arc between entities in the view shows a summary of method call activity between the entities.

The task required the students to implement displaying detailed information on those calls when the arc is double-clicked. The detailed information was to consist of class names mapped to the "caller" and "callee" abstract entities connected by the arc and was to be printed to standard output.

Figures 4.3 and 4.4 show the network of change tasks (boxes) related to study tasks A and B, respectively, along with the classes (ovals) linked with those recommendations. It is apparent from the figures that most, but not all, of the classes that needed to be modified in each task could be reached by following a sequence of recommendations of past modifications. These sequences are short for some of the classes, which can be reached from the top level of change tasks related to the assigned study task. Other classes are reached only after several stages of recommendations are followed, and sometimes a class necessary for a task's solution is not reachable through Hipikat. For example, class `CategorySummary` (lower left corner of Figure 4.3) was part of the solution for task A, and participants could find out about it from Hipikat in two steps: first by reaching change task "Reducing memory footprint...", recommended for its similarity to study task A, and then looking for files that implemented the change. On the other hand, none of Hipikat recommendations point to class `CelView` (in the middle of Figure 4.3, just below the top), so the participants had to find out about it on their own. We come back to these "maps" when presenting the results of the study, for illustration when participants refer in their comments to specific recommendations and software artifacts.

Hipikat client

Unlike the current implementation described in Chapter 3, the mock-up client used in this study was a stand-alone Java application (see Figure 4.5). The client allowed the user to browse the CVS repository, the online documentation, and the change task database. When the client displayed an artifact, it would issue a background request to the Hipikat server for items related to the displayed artifact, and would display a list of related artifacts returned in a side pane. The client also included a "bookshelf" where users could create notes as they worked and keep links to artifacts they accessed often.

4.1.2 Participants and procedures

As mentioned in Section 4.1.1, we conducted the study as a follow-on to an assignment in a graduate software engineering class. The purpose of the assignment was to introduce the students to program understanding tools. Students were grouped into pairs; each pair was asked to use the Hipikat mock-up for one change and one of Rigi [79], Ciao [23], or jRMTool [80] for the other. The order in which a pair had to perform each change task and the tool which it had to use were different for each pair.

The pair assignments were created by drawing a three-by-two-by-two matrix of non-Hipikat tools (Rigi, Chava, or jRMTool), task to do with that tool (A or B), and whether that task was to be done first or second. (The other task would be done using Hipikat.) Student teams were randomly

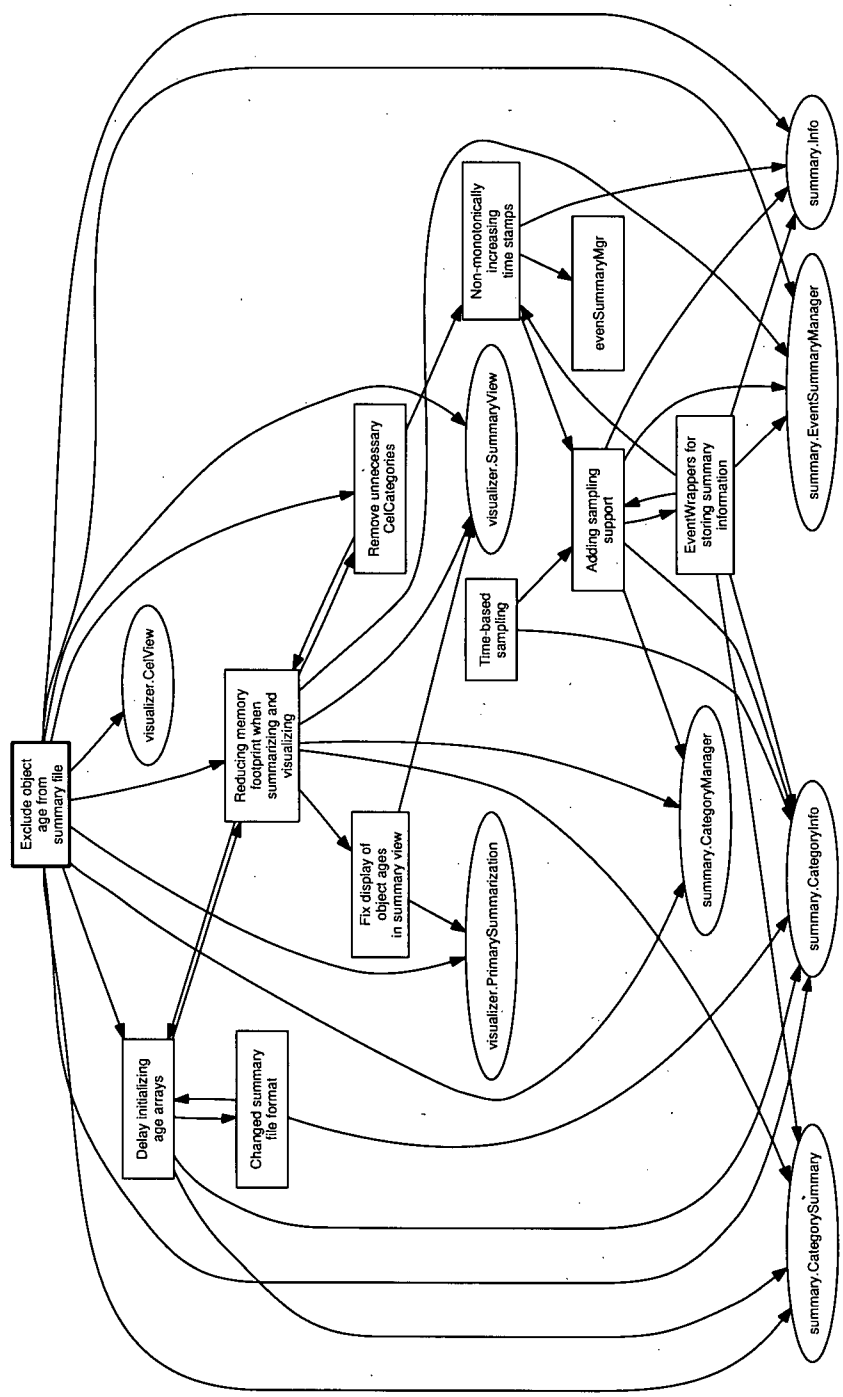

Figure 4.3: Related artifacts for task A (boxed in bold). Change tasks are represented as boxes, classes as ovals, and arrows point in the direction of the recommendation.

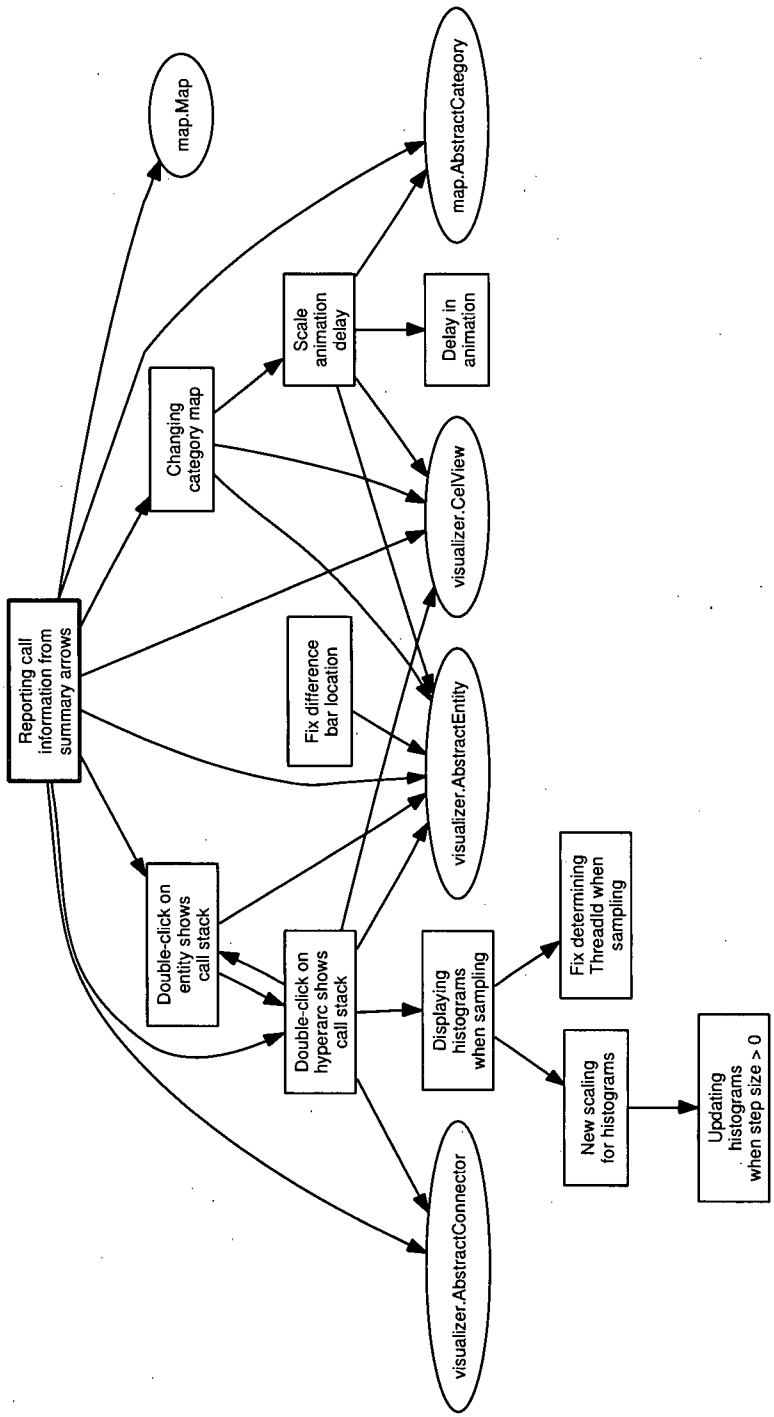

Figure 4.4: Related artifacts for task B (boxed in bold). Change tasks are represented as boxes, classes as ovals, and arrows point in the direction of the recommendation.

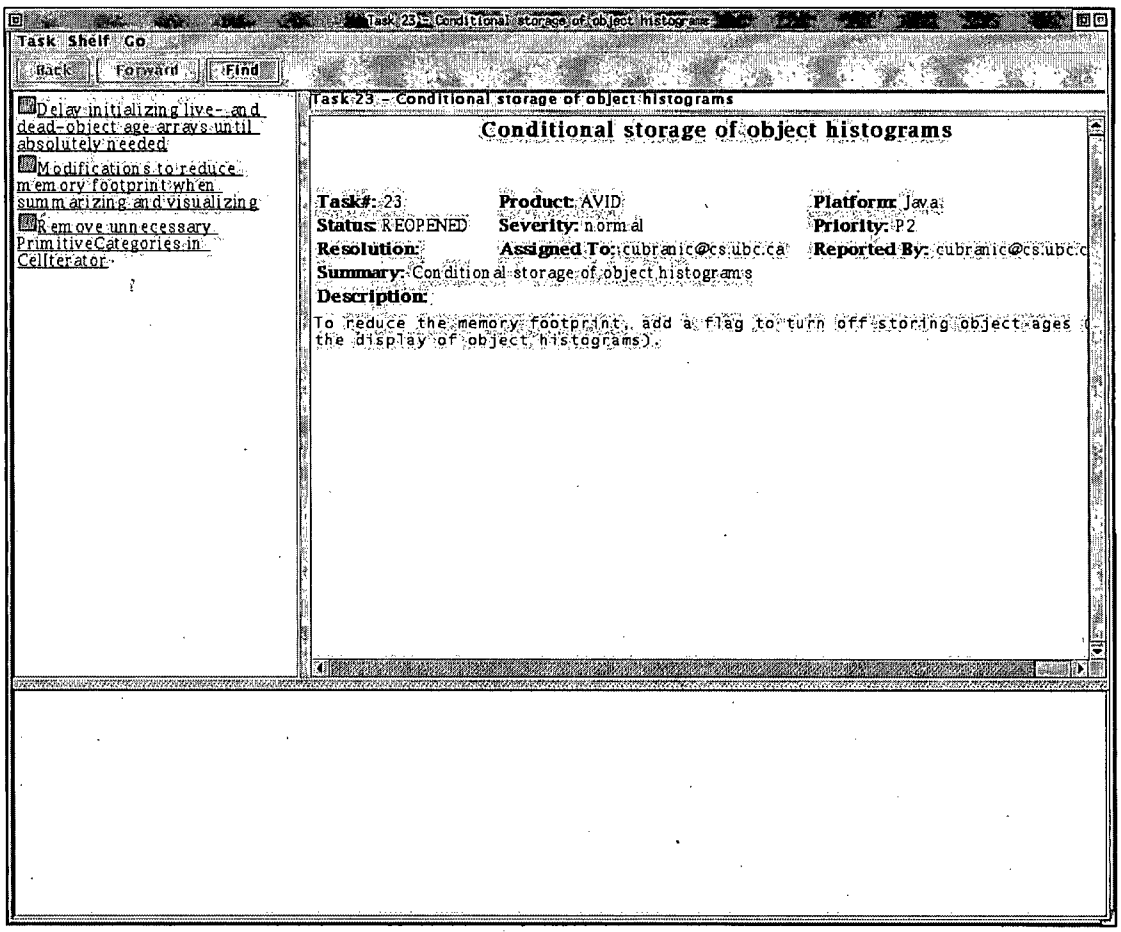

Figure 4.5: Hipikat client used in the study

assigned to one cell of the grid. There were twelve teams in the class, so no team had the same $(tool, task, order)$ assignment.

Earlier in the course, the class had two sessions in the lab where they had a chance to try out each of the four tools on small practice problems. During the assignment they were also provided with access to user guides for each tool.

Students had two weeks to complete the assignment. The deliverable was an assignment report describing their work process in each of the two tasks and comparing the two software understanding tools that they used.

Following the completion of the assignment, we invited students to participate in the study by sharing their assignment reports with us. Participation in the study was voluntary, and the identities of students who chose to share their assignment reports with us or to be interviewed about the assignment were kept anonymous to the class instructor until the semester ended.

Seven pairs (out of twelve in the class) agreed and gave us a copy of their assignment reports. We analyzed the reports for comments on Hipikat only, since our purpose in this study was not comparison to other software tools. Thirteen of the fourteen participating students had no previous knowledge of Avid. One worked the previous school year on porting Avid's user interface to a different UI library, and was familiar with most of the classes used in task B. (This participant was part of "pair 5" in quotations from assignment reports below. The participant did not take part in interviews.)

As a follow-up to the reports they shared with us, six students (out of fourteen) agreed to participate in an interview. We conducted the interviews individually, even for the two pairs where both students agreed to an interview. This was partly for the reasons of logistics, and partly to get the individual perspectives on the assignment, as opposed to the pair's consensus that was in the assignment report. While in one pair we did get opposing views of the team's dynamics, we did not find any contradictions with respect to tool use.

4.1.3 Results

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

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

> **Interviewee 2:** It helped really fast on the double-click [in task B]. We had the double-click in five minutes.

Recognizing useful recommendations

For the most part, the participants in our study were able to distinguish which suggestions were likely relevant, and which were the result of apparent similarity.

> **Interviewee 2:** [W]e didn't really go through the list in a "just search top-to-bottom" manner. We were like, "OK, Animator probably has something to do with the animated arcs, let's look [at it]".[1]

However, some pairs reported difficulty in assessing relevance without "wasting a lot of effort" investigating a suggestion (Pair 5). This was especially the case in the early stages of the exploration, when the participants would simply go down the list of the recommendations and examine them one by one:

> **Written comments from pair 5:** Until we examined the file HighLevelModel in the list of suggested artifacts, we wasted a lot of effort examining classes in [preceding recommendations in the list] suggested by Hipikat.

Another pair was more selective in their exploration and initially missed one of the most useful recommendations in task B until they really understood the change they were being asked to perform:

> **Interviewee 2:** Because we didn't understand the task [B], that [sic] when we saw "why does reloading the map [the top recommendation] have something to do with this?", because we didn't understand that. ... And when we understood the task and we actually looked at that [recommendation] and we saw, "OK, that's exactly [it]". [See Figure 4.6]

We also had reports of a participant pair missing a relevant suggestion because they lacked knowledge about the overall structure of the system, and realized the suggestion's relevance only once they had figured out the solution on their own:

> **Written comments from pair 7:** For the implementation of the change we ignored the existance [sic] of the AbstractQueryManager[2] although the Hipikat tool more or less directly pointed us to it through the change task for showing the call stack. Instead of using the AbstractQueryManager for the query we had to implement we put our implementation right in the detailedQuery function in the AbstractConnector. By doing this we violated the structure of the system.

That is, it does not help that the recommendation is objectively useful if the developer does not recognize it as such. The following two comments further illustrate this problem: they both talk about the same change task and the same set of recommendations, but reach exactly opposite conclusions:

[1] Note in Figure 4.4 that class Animator *was not* part of solution to task B.

[2] There is no class "AbstractQueryManager." It is likely that the report talks about "DetailedQueryManager," which was used by one of the classes implementing the fix for the related task, although it did not have to be part of the solution to task B.

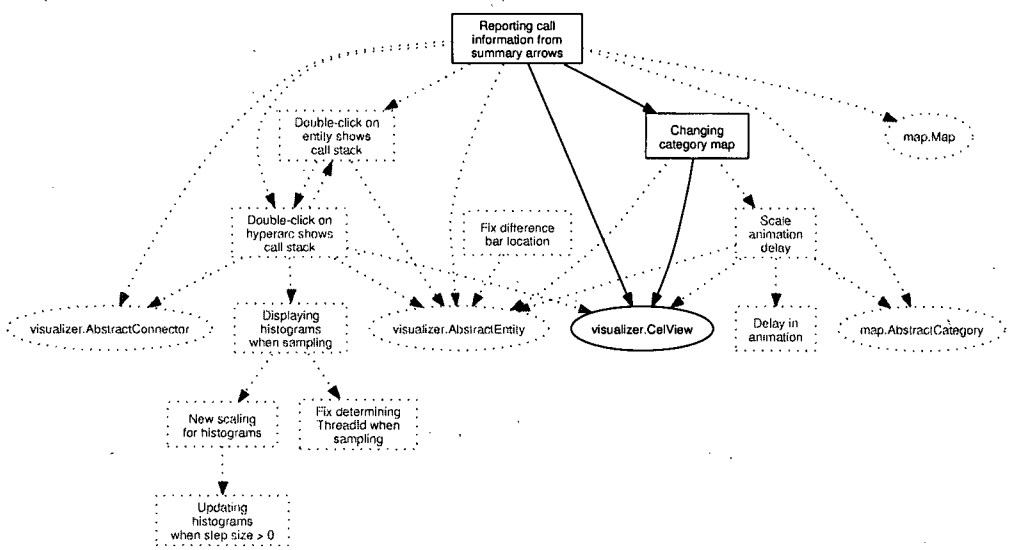

Figure 4.6: Recommendation trail for "reloading the map".

> **Written comment from pair 2:** Unfortunately, none of the change tasks previously recorded in Hipikat bore much resemblance to the change we were attempting aside from identifying a file...

and

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

Participants' confidence

A related challenge that study participants reported is applying the knowledge they learned from Hipikat recommendations to their task.

> **Interviewee 2:** [After seeing a recommendation that they knew could be reused in their task.] But we still weren't 100%, I wasn't convinced that—I was convinced of the fact that that was the easier way, to do it that way. But I still wasn't convinced that it was the better way.

That is, even when users realize that they can use the code from a recommendation in their own solution, it can be difficult to decide how much can be reused, what pieces are missing, or even whether it is the "right" way to implement the solution. Hipikat did not offer any assistance in this regard beyond providing the recommendations, but the study format also did not allow us to capture

how the participants handled this situation, which would be an essential step before we can consider how the tool could be improved in this regard.

User interface issues

The prototype of the Hipikat client-front end that we used in the study was designed as a code browser, rather than as an editor. Thus participants had to switch between two applications as they worked, and most considered it a drawback:

> **Written comments from pair 6:** Hipikat only shows the source code to the user. The user cannot modify the source in Hipikat. We have to open an editor to do the change task.

Perhaps because having to switch applications is distracting, the Hipikat client's code-browsing functionality was underused. The participants often preferred to use an editor even when only viewing the code, citing convenience, custom, or the added features that it offers:

> **Written comments from pair 7:** We used emacs obviously to edit the source code but also to read the source code as its syntax highlighting helps understanding the source a lot.

> **Written comments from pair 5:** The source repository viewer is not useful because Hipikat does not have a sophisticated text editor like "emacs".

Another consequence of this browser vs. editor division was that Hipikat showed the code of file revisions in the CVS repository, while editor showed local copies. Once those copies were changed, the two views diverged, which potentially was confusing:

> **Written comments from pair 3:** [Because] the source files Hipikat work with are the ones in the CVS repository, and we do not do any check-ins, ... our modifications on the source code could not be viewed ...

Additionally, the prototype's user interface, while simple and easy to use, was also limited in its power. For example, it did not have multiple windows which would allow viewing of more than one file at a time, and its "find" function was limited to searching within a single file. One user also commented that having the recommendations change constantly as he was browsing the repository was distracting. This behaviour could be annoying when he was exploring recommendations in a list, and the act of opening a recommended artifact caused the recommendation list to change, so he would have to use the "Back" button to restore the old list when he wanted to continue examining it.

Lastly, the interviews suggested having the reason for making a recommendation visible, including how relatively important it is:

> **Written comments from pair 5:** Since Hipikat's main goal is to provide a ranked list of suggested artifacts, a natural extension to Hipikat is to allow a search that provides sort[ed] list of matching occurrences by relevance.

4.1.4 Conclusion

The results of this study showed us that in some instances Hipikat can be a very useful tool. These instances support our first research claim, although there were also cases in which Hipikat was not helpful in the study task. Despite in some cases contradictory evidence (e.g., the two comments from participant pairs #2 and #4 on page 70), the study was an important step in refining our research ideas and the implementation of Hipikat, both the project memory and the client front-end.

The comments we collected from the class reports and during the interviews drove the subsequent changes to the tool to make it more usable and useful. The participant feedback made it clear that Hipikat client should not try to duplicate an editor's functionality; ideally, it should be integrated into an existing development environment. Furthermore, we decided to change the user interface from automatically suggesting related artifacts to making suggestions in response to a query from the user. We also included showing to the user the reason a recommendation was selected and the strength of its relation to the query artifact. Lastly, we narrowed down the artifact linkage heuristics to the present set (see Chapter 3), since those were the ones behind recommendations that seemed to be the most useful in this study based on the participants' comments.

Just as importantly, this study helped us define the questions we asked in the next research step, as well as the framework best suited to gather the kind of data in which we were interested. The results we collected in this stage showed that it was possible to make reasonable suggestions using the heuristics we had devised, and that those suggestions were (sometimes) useful to developers. However, we felt that by relying solely on post-hoc written reports and interviews, we were missing the *context* in which Hipikat was applied, something Shneiderman and Carroll pointed out is crucial to understanding how software tools are used [106]. We also missed *how* the information from Hipikat recommendations was used as the participants worked on the change tasks, except through the occasional glimpse:

> **Interviewee 2:** [When] we actually looked at that [recommendation] and we saw, "OK, that's exactly where we have to actually implement it, where we have to get all the information, put it into a data structure, and just print out that data structure when the arc is double-clicked."

We therefore decided we needed to observe and study the programmers' behaviour as they used Hipikat while working on software change tasks. We were influenced by Bowdidge's observational studies of StarDiagrams [13], but building on our experiences with the exploratory study, we decided to use a large open-source software system as the change target, and to draw study tasks from real modifications implemented in the target system between two milestone releases.

4.2 The Eclipse study

Following the results of the exploratory study, we decided to investigate more closely how Hipikat was used in order to evaluate its usefulness. Specifically, we asked 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? We wanted to

see if relevant examples Hipikat provided from the project's history can help the newcomer get started or to fill in the background information.

2. When and from which artifacts will newcomer developers who are working on a software change task query Hipikat? We were interested in the kinds of questions they ask and the answers they expect.

3. How will the newcomers evaluate Hipikat's recommendations and how can they utilize those recommendations in their tasks? We were interested in whether the way Hipikat's recommendations are presented is adequate and whether there are ways in which it could be improved to better support the developers in their change tasks.

An important factor influencing the design of the study was the need to have realistic participants working on realistic tasks. Firstly, we were interested in studying *newcomers*, not novices: our participants needed to have adequate programming experience in the programming language of the system under study, 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 with only a basic knowledge of a programming language and insufficient experience working on large software projects.

Secondly, we wanted to study our participants as they worked on tasks that were complex enough to challenge them and to require serious effort to understand the problem and come up with a solution. Specifically, we looked for tasks that would require a couple of hours to solve; otherwise there would not be much need for advanced software engineering tools, such as Hipikat. We also wanted tasks that were appropriate as an assignment for a project newcomer in a real project. The project we used in the study was Eclipse, an open-source software project initiated and actively supported by IBM. We considered mostly requests for new features in Eclipse, focusing in particular on those requests that had a visual or UI component to them because it made it easier for the participants to understand the task and test their solutions. Finally, the tasks had to allow for exploration and variation in the learning and problem-solving process of individual participants, but at the same time we wanted to be able to compare the solutions with each other and evaluate them for "correctness" and quality.

Given the questions asked in this research and the complexity of the tasks analyzed, we determined that a case study was again 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 product with that of ex-

perienced 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 with experienced team members.

4.2.1 Design

Eclipse is an extensible integrated development environment (IDE) that is written in Java and contains around a million lines of code. As is common for an open-source project, the development of Eclipse is conducted in a very transparent manner, with the full history of changes to the code, developer discussions, and problem reports publicly available. We selected as study change tasks two previously completed enhancement requests drawn from the Eclipse issue tracking database. 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, as described in Section 4.2.5. 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 Hipikat group memory on this copy.

The Eclipse team uses the Eclipse IDE itself for all development, so that was the natural choice of the development environment for the study (that is, all participants used the Eclipse IDE to make changes to its source code). All the information sources used by the project—the web-based documentation, issue-tracking system, source code repository, newsgroups, and mailing lists, as well as standard search engines used by the Eclipse project—were available to each participant, with the same applications to access them that they would normally use (e.g., Internet Explorer for viewing the web pages). Additionally, the newcomers had access to Hipikat, which is itself written as an Eclipse IDE plug-in, and thus was seamlessly integrated into their development environment.

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 order and learning effects.

Easy task

This modification request[3] described a need, when hovering over a particular point in an editor for Java source code, to display a breakpoint's properties in a pop-up window.[4] 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. (Figure 4.7 shows a screenshot illustrating the fully implemented request in use.)

[3]Request 6660 in the Eclipse problem report database

[4]A breakpoint is a debugging facility that suspends the execution of the program at a certain location in the code, enabling the developer to investigate the program's internal state.

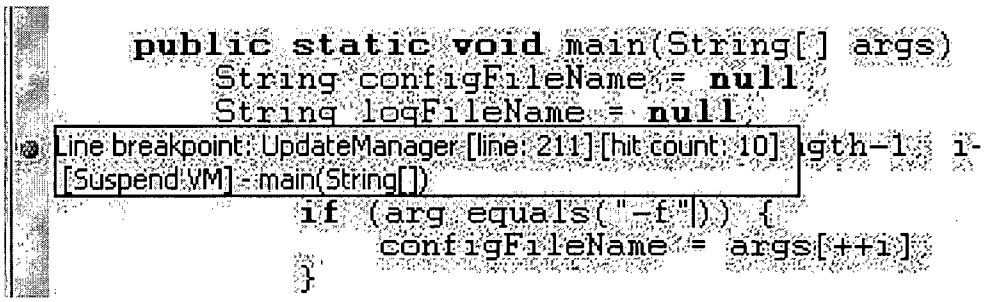

Figure 4.7: Breakpoint hover pop-up, as implemented in Eclipse 2.1. The pop-up indicates the breakpoint's line number, hit count, and the suspend policy, in addition to the class and method where it is located.

Difficult task

The second modification request[5] 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). In normal operation, the versioning context menu that comes up when the user right-clicks on a file in the IDE gives the user an option to, among other operations, *commit* a new version of a versioned file to the repository, or to *add* a new file to the versioning system, thus making it "committable" in the future. The modification request explained that all operations in this menu were incorrectly greyed-out (made unavailable for selection) when a mix of versioned and new files was selected—therefore the user would have to remember the state of files to ensure that only files with the same state were selected together for versioning operations or they could be neither committed nor added from the menu. A related problem noted in the same request was that committing a new file to the repository requires two steps: "adding" to mark it as a versioned file and then "committing." The request 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. The steps necessary in the solution to the task are shown in Figure 4.8.

Characterization of the two tasks

The categories "easy" and "difficult" are relative. Even the easy task was not trivial. The obvious way to go about solving it—looking for the mouse hover handler and working backwards—would lead the developer through some rather complicated code, having to understand multiple subsystems of Eclipse without really making progress on the task itself. A significant difference between the two tasks was in how much help the Hipikat recommendations provided towards the task: Hipikat provided at least one recommendation that makes it substantially easier to solve the "easy" task.

[5]Request 20982 in the Eclipse problem report database

1. Enable the "commit" option in the versioning menu even if there are new files in the selection.
2. If new files are present in the selection when the "commit" operation is chosen from the menu, notify the user and ask him whether to add them first.
   (a) If the user answered "yes," mark the new files for versioning and proceed to committing the entire selection.
   (b) If the user answered "no," do not mark the new files for versioning, and proceed to committing only the ones already versioned.
   (c) If the user answered "cancel," stop the whole process
3. If there are any files left to commit, proceed by asking the user for the check-in comment.
4. If the user entered the comment and pressed "OK," commit the files.
5. If the user pressed "cancel," do not commit and return.

Figure 4.8: The high-level solution to the difficult task.

Scope of change. The two tasks differed not only in the amount of code needed to implement a solution, but also in the extent to which the solution needed to interact with the rest of the system.

Easy: The scope of change to the system's source code was fairly isolated; it was located in only two classes and interacted with only a single other class in the system.

Difficult: Here too only a few classes needed to be changed, but the code in those classes interacted with a number of different subsystems: file management, versioning, and user interface.

Relevant Hipikat's recommendations. Participants relied on how high a recommendation was ranked in the response to the query response. For each task, we ensured that at least one of the recommendations provided by Hipikat in response to a query on the starting point—the task's problem report— was relevant. Determining that a recommendation was relevant depended upon how "obviously" similar its description (summary for a problem report or a CVS check-in comment) was to the current task, and how easy it was to evaluate the code in file revisions related to a suggested problem report for potential usefulness.

Easy: The top recommendation returned by Hipikat for the easy task was a previous enhancement request that was similar in description. The implementation code linked to this request was fairly straightforward to understand and coincided almost exactly with the code that had to be changed for this task.

Difficult: Hipikat recommendations had to be examined in more detail to accomplish the difficult task. The most useful recommendation (corresponding to the change that implemented the committing of new files in the "synchronization view") was not at the very top of the list as in the easy task, so the participants had to look at a number of recommendations before deciding which to investigate in detail to help them in the difficult task. Evaluating the various recommendations was more involved than in the easy task, because they were linked to implementation code that was more complex and spanned multiple files.

Learning from recommendations. Applying knowledge learned from the relevant Hipikat recommendation to solving the actual task depended on how difficult it was to understand the recommended code and how the code interacted with the rest of the system. Also important was how close the location of the recommended code was to that of the task's solution; if the recommended code was in a different subsystem, this included how hard it would be to "transplant" the code to the new location.

Easy: The recommended code was easy to understand. It simply puts the desired text for the hover message into a particular container object (the "marker").[6] Just as importantly, the recommended code was in the same file as the task's solution and used many of the same data structures.

Difficult: The related change reported by Hipikat was in a slightly different subsystem (the "synchronize view"). Although it could be used to determine a general approach for most of the solution, only small sections of the code could be reused directly because the data structures on which it operated were different.

4.2.2 Participants

Twelve volunteers participated in the study. The volunteers were paid an hourly honorarium for the time they spent in the study.

Eight volunteers were new to developing Eclipse, although some had used it as their Java development environment for 1 to 12 months and so had experience as users. All of the newcomer participants rated their familiarity with Java programming as at least "comfortable," and had experience working on large projects, including software management practices such as source code versioning and issue-tracking. Seven of the eight were graduate students and one was in the final term of his undergraduate degree.

The four expert participants were all cooperative work students in at least their second term at the IBM lab that is the leading contributor to the Eclipse project. The work term at the lab is four or eight months; consequently all expert participants had at least six months of experience developing and extending Eclipse, although their expertise was in different parts of the system than the selected enhancement requests touched. Therefore, the experts were familiar with the system's architecture

[6]There was no graphics handling in the recommended code because the container handles the display of the pop-up window, relieving the developer of all GUI considerations.

and the accepted ways of doing things, but still had to engage in information gathering to understand unfamiliar code.

4.2.3 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. To avoid interference with their regular jobs, the experts participated in the study on a weekend, since they did not need the training session.

Training session

Each of the eight newcomers underwent four hours of hands-on Eclipse training. The participants individually worked through three online tutorials that included frequent hands-on exercises applying the covered material. The participants worked on their own, but the researcher was present in the room to answer any questions.

The first training session covered the use of Eclipse to write Java programs in general. The tutorial took an hour and was based on the material in the *User's Guide* that is part of Eclipse's online help. Although four of the eight participants had previous experience using Eclipse (to work on their class assignments, for example), we required that all go through the tutorial to ensure the same basic knowledge of the environment's capabilities for writing, running, and debugging Java programs.

The next two hours covered programming and extending Eclipse itself. This material was based on the online *Programmer's Guide* that comes with the Eclipse distribution. It is reasonable to assume that "real-world" Eclipse newcomers would have gone through these online guides, because the guides were the only introduction to Eclipse available until third-party books were published in the Summer of 2003, while the version of Eclipse and the change tasks used in the study dated from the Summer of 2002.

The last hour of the instruction covered Hipikat, from its design and features to a walk-through of a sample session using Hipikat to work on an Eclipse problem report. This part ended with an open-ended exercise where the participants were asked to complete a bug fix using Hipikat to give them some experience using the tool in a less structured format.

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.

Programming session

The programming session was divided into two parts, one for each change task. The maximum time to allowed for each task was fixed at two and a half hours.

Each part started off by randomly assigning one of the change tasks to the participant. The participant was then seated at a computer workstation displaying a window with the study instructions (Appendix E) and running the Eclipse IDE with the full source of Eclipse 2.0 in the workspace.

The instructions directed the participant to the description of the change task in the study copy of Eclipse's Bugzilla database. The instructions asked the participant to read and understand the task. Once the requested feature was understood, the participant was instructed to notify the experimenter (that is, the author of this dissertation) and explain the feature to him.

The participant then moved onto the next section of the instructions (Appendix E.1.1), which asked the participant to prepare a plan of the change. The participants were free to use any available Eclipse tools to understand the code and to plan their change, but we requested that they complete the plan and describe it to the experimenter before proceeding with implementing the change. The format of the plan was left to each participant, and the level of required detail left flexible, perhaps only including the broad outline of the approach and the list of files that would need to change. The use of Hipikat was explicitly encouraged in the instructions to the newcomers. If a participant did not come up with a plan by the end of the first hour, we conducted a progress interview to see what the participant had been working on. Participants who got stuck during the planning stage were given a small hint if they were entirely off the solution's track. (This is comparable to advice they may have got on an Eclipse newsgroup, for example.)

Once the participant notified us that the change plan was completed, we conducted a semi-structured 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. Following the plan interview, the participant went on to the next section of the instructions (Appendix E.2), which directed him or her to start implementing the change. When the participant notified us that the change was implemented, or if the session time limit was reached, we conducted another semi-structured 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 [38]. We also asked the participants about available tools and information that were useful or not useful, as well as those that would have been useful had they been available.

4.2.4 Data

We used screen capture software (Camtasia by TechSmith) to record the participants' actions while working on the change plan and its implementation. We also instrumented Hipikat to record all queries in a file, although this information could have been obtained from the screen recordings by manual means.

The participants submitted their change plans at the end of each change task. In all instance but one these plans were created electronically, in a word processor. We later printed a hard copy and used that in the subsequent analysis. One participants wrote the plan by hand and submitted his notes.

The implementation of the change tasks was obtained by printing the source files that a participant modified in the course of each change task. These files were printed after both change tasks were completed. Because no participant modified the same file in both tasks, there was no need to collect the changes after each task.

4.2.5 Analysis

All recordings we made were first manually transcribed: interview tapes to text, and screen recordings to maps of each participant's exploration. The maps were made by marking when each artifact (bug report, source file, revision, web page) was viewed for the first time and the way in which it was reached. When two artifacts were logically related as part of the same "exploration path" (for example, seeing the use of an identifier in one file, and jumping to its definition in another file), we connected them with a directed edge, forming a directed tree of such paths. See Figure 4.9 for an example.[7]

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. These criteria—shown in Tables 4.2 and 4.4—are 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. Lastly, we required that the added code be readable and maintainable, and that it follow the Eclipse team's coding practices.

We manually evaluated participants' solutions based on the criteria. When a criterion was missing from the solution, we checked for it in a participant's written change plan, and in the comments the participant made during the change plan interview. (This is noted in the presentation of results, but did not count as an error.) We were interested in the change plans in addition to the solutions because we wanted to account for all the information participants discovered, even if the solution was partly incomplete because of our arbitrary time limit for work on the task.

4.2.6 Results

In this section we present the results of our analysis. We first focus on the participants' performance in each of the two tasks. We describe the patterns in their solutions and compare the newcomers with the experts. Next, we look at their process: how the newcomers accessed Hipikat, and how they evaluated and used Hipikat's recommendations in their work.

We present our analysis of performance for the easy task first, but note that the order in which the tasks were done did not seem to make any difference in the performance across participants. For example, the scores on the easy task for those newcomers that worked on that task first were: 95%, 90%, 85%, and 40%. Newcomers who worked on the easy task second scored: 95%, 95%, 70%, and 65%. The difference between the means of the two groups is three percentage points (78% vs. 81%), which could hardly be considered significant even in the non-statistically rigorous sense.

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. The experts solved the task much faster, while most newcomers took all of or close to the full time available (average an hour and a quarter for the

[7] We used Excel so we could organize the entries in a grid for easy viewing of the timeline.

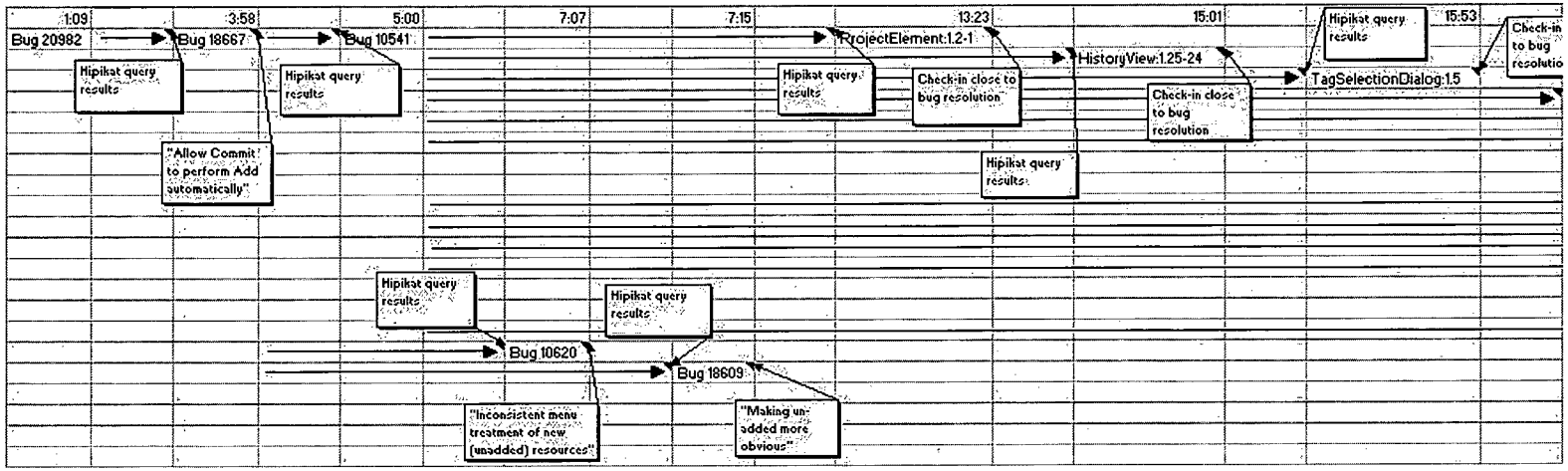

Figure 4.9: A portion of a newcomer's exploration map. Time flows to the right, and the top row shows the exact time from the start of the task when the artifact in that column was opened.

The exploration starts at the top left corner, with the opening of bug 20982 (i.e., this is the difficult task). The next artifact that the participant opens is bug 18667, because it is the next one going to the right. It was reached directly from bug 20982 because the two are connected with an arrow. Comment attached to the arrow indicates that the traversal was the result of a Hipikat query. Comment attached to bug 18667 shows its title (i.e., the "summary" in the Bugzilla terminology).

Bug 10541 is opened five minutes into the task in a similar manner, only this time as a result of a Hipikat query on bug 18667, as are bugs 10620 at 7'07" and 18609 at 7'15" (note that the arrows leading to all three of those bugs start in 18677's column).

Then the participant goes back to bug 10541, makes a Hipikat query on it, and from the returned list of recommendations opens the diff view for version 1.2 of file ProjectElement.java at 13'23". The reason for recommending this version is shown in the comment attached to it: "Check-in close to bug resolution."

The exploration continues in a similar manner in the rest of this example, with all the other artifacts visible in the figure reached from Hipikat's recommendations for the query on bug 10541.

experts vs. two hours for the newcomers, see Table 4.1 for full details). However the whole picture is more complex. All but one of the newcomers found the right files for the change quickly (using Hipikat's recommendation), but then took much longer than the experts to understand the intricacies of the code.

Subject	**Time (minutes)**
Alfred[a]	99
Bill	45
Cecilia	56
Dennis	93
Average	**73**
Emil	150
Frank	79
George	119
Harry	67
Ida	150
John	150
Keith	128
Louis	134
Average	**122**

Table 4.1: Time taken to implement the solution for the easy task. The participants are grouped into experts and newcomers.

[a] All names have been changed to protect participants' anonymity. Names were chosen arbitrarily so that each starts with a different letter, but reflect the participant's gender.

Although all solutions presented the pop-up with breakpoint properties as requested, many participants did not handle the special cases properly (e.g., updating the text in the hover after the breakpoint's properties were changed in the properties dialog), which introduced bugs into their solution. Surprisingly, this was even more the case among the experts, where only one of the four participants (25%) 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%) handled this correctly, and two more (for a total of 63%) 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). See Table 4.2 for the full list of the correctness criteria used in this task, and Table 4.3 for a summary of the correctness of solutions for all participants.

The faulty solutions focused on a particular class, *JavaLineBreakpoint*, and missed consideration of that class's superclass *JavaBreakpoint*, which was responsible for updating some of the breakpoint properties. The faulty solutions thus missed some method inheritance interactions, so that the hover text was not updated when properties were changed through the JavaBreakpoint superclass. An examination of the plans created by the expert participants who failed to handle these

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

Style: Modifications should respect existing architecture and coding style.

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

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

- Method breakpoint hover is correct
- Watchpoint hover is correct

Table 4.2: Easy task correctness criteria.

updates shows that all of them talked exclusively about the concrete subclass. In a way this is not surprising because that is where the bulk of the change was located, and it was probably so deceptively simple that they did not investigate all of its implications.

The relatively high correctness of the newcomers' solutions cannot be explained entirely by the longer amount of time that they took to finish the task. They were not taking much longer to come up with their change plans, yet even there both classes were regularly mentioned. 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 inserted by the identification matchers. This connection was recognized by 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.

Another example showing that newcomers can learn good practices implicitly recorded in past solutions was the way they realized that they should "externalize" all text messages in their code so that the application can easily be internationalized by changing a few properties files that came with it, rather than the source code. This approach was not something that was explicitly covered in the tutorials, but became obvious in Hipikat's recommendation, which showed that whenever some changes involved adding text messages, those messages were being externalized in the properties

Subject	Hover			Update			Style		Subcl.		Correct
Alfred	+	+	+	+	+	+	+	+	+	$\diamond^{h}$	9.5 (95%)
Bill	$\diamond^{l}$	+	+	$\diamond^{c}$	+	+	–	+	+	$\diamond^{h}$	7.5 (75%)
Cecilia	$\diamond^{l}$	+	+	$\diamond^{c}$	–	+	$+^{p}$	–	+	–	6 (60%)
Dennis	$\diamond^{b}$	+	+	$\diamond^{h}$	+	–	–	–	+	$\diamond^{h}$	5.5 (55%)
Average											**7.1 (71%)**
Emil	+	+	+	+	+	+	+	+	+	$\diamond^{h}$	9.5 (95%)
John	+	+	+	+	+	+	–	$+^{p}$	+	$\diamond^{h}$	9 (90%)
Ida	–	+	$\diamond^{a}$	$\diamond^{a,s}$	$\diamond^{a}$	$\diamond^{a}$	–	–	$\diamond^{a}$	$\diamond^{a}$	3 (30%)
Keith	+	+	+	$+^{s}$	+	+	$+^{s}$	–	+	$\diamond^{h}$	8.5 (85%)
Frank	+	+	+	+	+	+	+	+	+	$\diamond^{h}$	9.5 (95%)
George	+	+	+	$+^{s}$	+	–	–	–	+	$\diamond^{h}$	6.5 (65%)
Harry	+	+	+	$+^{s}$	+	+	$+^{s}$	+	+	$\diamond^{h}$	9.5 (95%)
Louis	+	+	+	$+^{s}$	$\diamond^{o}$	$\diamond^{d}$	–	$\diamond^{e}$	+	$\diamond^{h}$	7 (70%)
Average											**7.8 (78%)**

[a] sets hover for all vertical ruler marks, not just breakpoints
[b] plan mentions only JavaBreakpoint
[c] not updated until the condition is changed
[d] duplicate line number if both hit count and condition are enabled
[e] strings are not always externalized
[h] not updated until the hit count is changed
[l] plan mentions only JavaLineBreakpoint
[o] hit count disappears if the condition is off
[p] correct in the plan only
[s] suspend policy not implemented

Table 4.3: Participants' performance on the easy task. The participants are grouped into experts, newcomers who worked on this task first, and newcomers who worked on this task second. Symbols: + – correct; - – incorrect; ◇ – partially correct.

file. Three of the eight newcomers externalized their strings, and two more noted the practice and said that they would probably do the same in their code before releasing it as a finished product. On the other hand, all of the four experts were aware of this practice although only two actually externalized their strings, while the other two admitted that they would have done so before releasing their code to the rest of their team.

Difficult task solutions

The participants were less successful in solving this task, with the average score of 8.5 out of 15 (57%). While the two groups have virtually the same group average score (8.4 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%) 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 the average score of 10.7 (71%).

In the newcomer group, three of the eight participants (38%) managed to implement all of the basic requirements (average score 11.7, or 78%). 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 last 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 that respect, all of the newcomers got farther than the unsuccessful expert, since even their incomplete solutions were on the right track.

See Table 4.4 for the full list of the correctness criteria used in this task, and Table 4.5 for a summary of the correctness of solutions for all participants.

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 desirable 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. (We come back to this issue and consider how the presentation of such recommendations could improved in Section 5.2.1.)

Furthermore, because the code in Hipikat's recommendation was in a different subsystem, in which it was assumed that the contents of the directories were already available, the newcomers

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

Table 4.4: Difficult task correctness criteria.

Subject	Menu			Detect new			Msg. dialog				Cmt.		Style			Correct
Alfred	+	+	–	+	+	–	+	+	$\diamond^d$	+	+	+	–	$\diamond^f$	–	10 (67%)
Bill	+	+	–	+	+	–	+	+	+	+	+	+	+	$\diamond^f$	–	11.5 (77%)
Cecilia	+	$\diamond^p$	–	–	–	–	–	–	–	–	–	–	–	–	–	1.5 (10%)
Dennis	+	$\diamond^n$	–	+	+	–	+	+	$\diamond^d$	+	+	+	+	$\diamond^f$	–	10.5 (70%)
Average																**8.4 (56%)**
Emil	+	$-^s$	–	+	$-^s$	–	+	+	+	+	–	–	+	$\diamond^f$	–	7.5 (50%)
John	+	+	–	+	+	–	+	–	+	+	+	+	+	$\diamond^f$	–	10.5 (70%)
Ida	+	+	–	+	+	–	+	–	–	+	+	–	+	$\diamond^f$	–	8.5 (57%)
Keith	+	+	–	+	+	$\diamond^a$	+	+	+	+	+	+	+	$\diamond^f$	–	12 (80%)
Frank	+	+	–	+	+	–	+	$\diamond^c$	+	+	+	+	+	$\diamond^f$	$\diamond^r$	11.5 (77%)
George	+	$\diamond^p$	–	+	$-^s$	–	+	–	–	–	–	–	+	$\diamond^f$	–	5 (33%)
Harry	+	+	–	+	+	–	+	+	+	+	+	+	+	$\diamond^f$	–	11.5 (77%)
Louis	+	–	–	+	–	–	–	–	–	–	–	–	–	–	–	2 (13%)
Average																**8.6 (57%)**

[a] Does not actually add the children to the selected list.
[c] No add when the check-in comment dialog is cancelled.
[d] will show the check-in comment dialog even if no files will be committed
[f] uses ForceCommitSync's externalized string.
[p] Does not work if the parent is not a CVSFolder.
[r] Visitor does not recurse, although noted a question about it.
[s] Tried using CVSSyncSet (causes runtime cast exception).

Table 4.5: Participants' performance on the difficult task. The participants are grouped into experts, newcomers who worked on this task first, and newcomers who worked on this task second. Symbols: + – correct; - – incorrect; ◇ – partially correct.

overlooked that this was something they would have to do themselves if they were to reuse it in their subsystem. This raises an interesting question of the kind of expectations that the newcomers had for Hipikat's suggestions, something which we consider in more detail in our later discussion (Section 4.2.7).

The expert participants did not look at problem reports other than the one describing the assigned study task, and so they never saw the above-mentioned discussion. Still, they used Eclipse daily in their work, so it was a little surprising that they forgot about the behaviour in the "Synchronize view" (particularly because that view was specifically mentioned in the assigned task description) and did not parallel its functionality in their solutions.

Instead, during interviews some experts said that they did not consider the directory special case at all; those who did consider it ended up concluding that the commit operation should not work that way. This conclusion diverges from the project consensus on the behaviour of the commit operation. If the requested feature had really been implemented this way in Eclipse, users of the system would have probably been confused with inconsistent outcomes of the commit operation depending on the point in the user interface at which it was invoked.

Accessing Hipikat

We used the exploration maps to try to find patterns in how and when Hipikat was queried and which recommended artifacts were accessed. 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.[8] In each case, one of the queries—usually the very first one, except as noted below—was on the problem report that describes the assigned task, so in the easy task in particular it did not take very long for the participants to find the information that they wanted from Hipikat.

Almost all queries were made within the first hour, especially when a participant was successful in formulating a solution plan. (Slightly less than 20% of all queries were made after the first hour, or 0.7 and 1.1 query in the easy and difficult tasks, respectively.) Once a participant knew the file(s) to be changed and had determined a general plan of how to do 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.

This is particularly obvious in the easy task, where four of the eight newcomers needed only two queries. Their first query, on the problem report assigned in the task, led to a very similar problem report which was at the top of Hipikat's recommendations and was marked fixed. Querying on that report led to the file revisions implementing its features, which pointed out the classes involved in the change and highlighted the code that was added. At that point, the participants would switch to viewing the source code of these classes and to using other Eclipse tools to understand how that code interacts with other parts of the system. Other participants who successfully solved the easy task also found that same top recommendation and eventually used it in their solutions. However

[8]Figures 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 used as an alternate query "history" mechanism, we did not count them when calculating the averages.

they made one or more other queries, probably to get a better feel for the code before plunging in to understand it more fully.

The participants queried Hipikat predominantly in the two-step sequence just described: find an interesting-looking problem report, then check to see if it has any associated file revisions that look like they could be relevant to the task, either as an example of the API usage or as a potential source of code to reuse. (This is not surprising because it was the main interaction technique shown to participants in their Hipikat tutorial.) Artifacts other than problem reports and file revisions were hardly ever considered, except in two situations. At the very beginning of the task, participants looked for explanations of some of the terms in the task description (e.g., "hover", "suspend VM"), not only in problem reports but also in the suggested web pages (that is, in online design documents and reference manuals). Second, when participants felt that the recommended problem reports were not giving sufficient information (or were suggesting file revisions containing code that was either irrelevant to the task or too hard to understand), they tried looking at recommended news articles for possible clues. This was only a temporary change of tactic, because the newsgroups in the Eclipse project are not used to discuss development issues. Therefore, news articles were not helpful in either assigned task, so participants who looked at them soon returned to their initial search strategy.

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 sub-goals. The following quote from one newcomer discussing the easy task illustrates this goal-driven behaviour:

> So, my first goal is to find out how the hover behaviour is implemented. ... the second thing was to find out where the line number, where I can get the line number at that stage, so I can add it in. (Harry)

The most important thing was to find code relevant to the current sub-goal. Of course, finding the right piece of code in a system containing hundreds of thousands of lines can be like searching for a needle in the proverbial haystack. Even using more sophisticated search strategies—like starting from a recognizable entry point and tracing backwards through call and inheritance graphs—could be likened to using a length of thread to find an exit from a labyrinth. We observed two of our experts struggle with the easy task using just such a strategy. They eventually succeeded in their search because of their experience. On the other hand, the newcomer who failed to solve the same task used much the same strategy, but got hopelessly bogged down trying to trace her way through the call and inheritance graphs. After much frustration, and despite several hints from the author observing her work, the participant found the code in the UI subsystem that displays pop-up hovers and forced her solution there. The solution displayed the line number in all hover pop-ups, not just for breakpoints (a similar mechanism is used to signal syntax errors in source code editors and to insert user-defined bookmarks), and violated architectural boundaries of the system.

The alternative approach, which we saw among 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 explore the first two points in the remainder of this section, and leave the third for the discussion (Section 4.2.7), when we explore some of its more general implications for learning from group memories.

Recognizing useful recommendations We found that the crucial first condition to recognize 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. In the case of such a problem report, the participants would then, via another Hipikat query, move to investigate the associated file revisions.

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.

The difficult task provides an excellent illustration of this behaviour. There, the initial Hipikat recommendation list contained two related problem reports near the top of the list. Both of them pointed to the same set of files, although the revisions associated with the one lower in the list actually highlighted code that could be directly reused to implement two of the task's basic requirements. In contrast, the higher-ranked recommendation was only useful in pointing to the right classes, but its implementation was more complicated and in the end not as useful for the assigned task. And yet, only two of the eight newcomers ever looked at the code implementing the fix for the lower-ranked, but in reality more relevant, problem report. Instead, the higher-ranked recommendation was "close enough." Its description was similar enough to the assigned task, and its implementation involved code that looked promising enough, so the participants were apparently reluctant to spend any more effort looking for something better. In the end, this course of action was successful, although it almost certainly took longer than following the lower-ranked recommendation. Given the uncertainty whether something better existed, it was not an unreasonable course to take.

Understanding recommendations In some cases, for instance in the top recommendation in the easy task, the code in 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. We will discuss the problem in more detail in the discussion (Section 4.2.7), including potential avenues for alleviating it. 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.

4.2.7 Discussion

In this study, the examples of previous changes provided by Hipikat were helpful to newcomers working on the two change tasks. The recommendations were used as pointers to snippets of code that could be reused in the assigned tasks and as indicators of starting points from which to explore and understand the system. Without such help, it is hard for a newcomer to a project to even know where to begin:

> before I actually saw the results Hipikat [[unclear]], I wondered how I would trace where the hover behaviour was coming from, and—and really I had no idea how that stuff is implemented. ... I mean, I can't, I don't know even if I would have gotten to it. I might have done some search on breakpoints, maybe that would have gotten to [[unclear]]. ... No I guess the breakpoint it doesn't actually implement IMarker. I'm not sure. Certainly wouldn't have been as easy. (Harry)

Moreover, the study also made it apparent that the recommendations provided were not used to gain wider understanding of the code, or of the design rationale behind it. For instance, although the problem reports recommended by Hipikat included developer discussions, such as design decisions and implementation trade-offs, it seems that the reports were read mostly to evaluate their closeness to the task at hand. Otherwise, the participants arguably should have noticed that the problem report used as a basis for most solutions in the difficult task contained a discussion of what to do with new files in subdirectories during a versioning operation in the "Synchronize view." However, none of the participants considered checking for this condition.

One possible reason for this oversight is that the study participants were focusing on implementing some basic functionality first, given the study's time limits, and leaving the improvements for later. Arguably, this is true in general: programmers and professionals are *always* under time pressure and driven to fix just the immediate problem. The question is whether project histories will tend to be used mostly as a source of "shallowly understood" examples, superficially knowing the functioning of a piece of code without understanding the pre-conditions that it expects when it is called or the results that it returns. As we saw in the difficult task, the consequences of such a

strategy are implementing a quick fix without deeper analysis, which may introduce subtle bugs and other problems in the future. This is a concern that deserves further study, but with users who have a real and long-term involvement in the project, because those users will have a stronger incentive to gain deeper understanding of the recommendations provided by a project history recommender, such as Hipikat.

On the other hand, it is possible that it was precisely Hipikat which allowed the study participants to solve the tasks without gaining a deeper understanding of the system, by making it easy to "lift" the code from recommendations. Rosson and Carroll [97] have observed this approach by developers. Developers naturally engage in an *as needed* comprehension strategy because it is the only strategy that is feasible given time constraints and limited information. Since Hipikat makes a wider range of information available, we expect that at times when more thorough comprehension is necessary, it would be a feasible action to take.

We should also note that the way the discussion was organized within the problem report—with little structure and no highlighting of the important parts and conclusions—made it too difficult to follow by someone who is just trying to get the overall picture. This is precisely the kind of situation that design intent systems attempt to address, but these systems are impractical to apply to the hundreds of problem reports that a large project, such as Eclipse, handles each week. It is possible that a more appropriate compromise can be found between the formal notations proposed by most design intent systems, and the simple chronological sequence of comments used in most issue tracking systems today.

When useful recommendations go unrecognized. The newcomer participant who was unsuccessful in the easy task provides an interesting lesson to consider: even the best examples are useless if the developer does not recognize them as such. As we already described, the top recommendation provided by Hipikat was a problem report which could hardly have been more similar to the assigned task, and which was linked to code that was fairly easy to understand and very helpful in solving the task. And yet, this participant looked at the recommendation and discarded it as irrelevant. It is hard to say what could have been changed in Hipikat that would have helped in this case, or to prevent it from happening again. It may be simply an unavoidable risk faced by newcomer developers working on an unfamiliar software project. It should, however, be less likely for a developer who has a larger stake in completing the change task, and who is willing to investigate more fully even recommendations that at first look seem unpromising. (A more careful investigation of recommendations could also be expected as users gain experience with Hipikat and come to rely on it more for providing useful recommendations.)

Understanding a recommendation's context. At least some participants who used Hipikat stated in interviews that they had assumed that the code they were using as a template in their difficult task solutions would automatically handle the subdirectory special case that was already described. They had misunderstood the *context* of the recommended code and how it translates to the new context in which they were developing. In the original context, the subdirectories were already being handled by the caller—something that was not true in the new context. Although *reuse of uses* has already

been studied by Rosson and Carroll [97], understanding the *usage context* during reuse of Hipikat's recommendations poses additional problems to the developer. Developers studied by Rosson and Carroll reached the code they were reusing by exploring the source on their own, building a mental model of the context as they went. Developers in our study using Hipikat had to build their mental model from the recommendation outward. This exploration was fairly "breadth-first", and so it is possible that it was not sufficiently "deep" to establish the right usage context.

We believe that some form of visualization could make it easier to understand the provided example in the context of the larger system, which may help alleviate this problem. In particular, the relationship between modifications in a set of versions, and to the other modules, could be made clearer by presenting it as a graph of methods and *use* relationships, rather than a series of source files diffs.[9] Otherwise, Hipikat users are faced with the situation in which, as one of the study participants described it, "everything is in drawers and you open one drawer at a time and look inside." (Louis)

Understanding a past context. Establishing the correct context becomes even more difficult when a recommendation is from a version of the system so far in the past that the API or, even more problematic, usage conventions have changed and are not valid in the current system. Detecting this situation automatically—thereby never making such recommendations in the first place—is unlikely. However, it is precisely the newcomer developer who will be most challenged to realize that parts of recommendations may not apply any more. This challenge could potentially be diminished by visual indication of discrepancies between the recommendation's and present code's structure, although it will not necessarily solve more semantic problems, such as the correct conditions which need to exist before a certain call should be made, for example.

4.2.8 Threats to the study validity

When designing our user study, we faced interesting methodological challenges and our choices reflect both practical and theoretical constraints. The lack of universally accepted measures for code quality and the fact that there were many potential variations in how the study tasks could have been done meant that we had to design our own correctness criteria that were more high-level and had the flexibility to accommodate a range of solution. Similarly, there are few meaningful measures of program comprehension—or of the program comprehension *process*—so conducting a qualitative analysis of comprehension issues was the only realistic option. Also, as we've already discussed above, we decided to rely largely on observation to answer the *how* and *when* Hipikat is used.

There were other logistical constraints that influenced our design. Although we wanted realistically difficult tasks, we couldn't require that study participants spend a week working on them full-time—they had to be doable in a day. Initially, we had allocated four hours per task, but then

[9]This problem is exacerbated by the fact that the Eclipse user interface arranges file editors using a "tab" metaphor, so that only the current one is visible (see Figure 1.10, for an example). While it is possible to get around it by having multiple Eclipse windows opened, it is cumbersome, and users rarely do it.

pilot testing showed that fatigue, compounded with frustration, was becoming a serious factor as the time progressed and that most of the interesting events happened in the first couple of hours. Consequently, we shortened the time allowed for each task to two and a half hours, which turned out to be sufficient to complete the exploration stage—which is when Hipikat is used—and to devise most, if not all of the implementation of the solution.

These methodological choices should be kept in mind when evaluating the threats to the study validity. Commonly, these threats are classified as follows:

Construct validity Construct validity refers to whether the study effectively measures what it is intended to measure;

Internal validity Internal validity refers to how a causal relationship is established to argue about a theory from the data;

Reliability Reliability refers to the degree to which someone analyzing the data would conclude the same results;

External validity External validity refers to the generalizability of the results of a study.

Construct validity

We address construct validity in two ways. First, we devised the criteria for evaluating the correctness of the participant solutions by closely following the solution implemented and adopted in reality by the Eclipse team for the release 2.1 of the product. The style-based criteria, "externalizing strings used in user messages" and "avoiding additional coupling," are seemingly less objective, since they are not based on satisfying a predetermined set of functional requirements like the other correctness criteria that we use. However, that does not mean that they are arbitrary. Externalizing strings is the policy adopted by the Eclipse project to ease localization of the software, and could be seen in the examples provided by Hipikat. Avoiding additional coupling between components is recognized in software engineering community as a "good" development practice. Given these criteria, we believe that it was possible to evaluate the correctness of the solutions quite objectively simply by comparing the code against the list of checkpoints, as in Tables 4.3 and 4.5.

Second, we collected the data from multiple sources to the extent possible: we videotaped the development process, interviewed the participants at multiple points in the study, and asked them to write down the plan of their changes before they started modifying the code. Each of the sources was useful to complement the others. For example, videotapes record exactly what was being done and when, but the answer "why" it was being done remains a guess unless it is corroborated by the participant in an interview. Interviews, on the other hand, are in danger of participant's post-hoc rationalizing of his or her actions, but this threat can be reduced by checking the participant's statements about what they were doing against what was actually recorded.

Internal validity

Internal validity of this study could be questioned primarily on the grounds of comparing the performance of newcomers using Hipikat to that of experts using their usual development tools. In the terminology of experimental design, the experts were not a true control group, because even though they did not use Hipikat, they also differed from the newcomer group in their experience with Eclipse. This difference means that the causal link between using Hipikat and performance is less clear, because it may be obscured by other factors, such of length of experience. However, given that newcomers outperformed experts on the correctness criteria, we feel that we have a pretty strong support for the argument that it was Hipikat that made the difference. This support may not be in a statistically significant form, but there should be no other reason to believe that programmers with only half a day of instruction on a million-line software system could do as well or better modifying it as those who have had a year of experience working on the project.

The value of even having the expert group is that thanks to it we were really able to see what could be done (and how) by an experienced member of the project in the time allotted for the study. We knew what the "correct" solution was, but we did not know how it had been reached. The experts' performance gave us only a rough approximation, but is none the less valuable and puts the newcomers' performance in the study into a better perspective.

Reliability

We attempted to ensure the reliability of our study with respect to data collection by having detailed procedures for conducting the study and gathering the data. Also, we used video- and audio-recordings rather than field notes, so that coding and analysis could be done from the original sources, rather than notes which have a layer of interpretation already build in.

With regards to whether another researcher evaluating the same data would reach the same conclusions, the main issue here is that we evolved our qualitative data analysis as we progressed. We did not attempt to collect a measure such as intercoder reliability [10] primarily because of the time commitment necessary to train another analyst and for that person to perform analysis sufficient to calculate the intercoder reliability. We note, however, that a significant part of the analysis was rather objective, such as checking whether a participant's solution implements the correctness criteria. The exploration maps were also constructed using very simple marker events in the video-recordings—specifically, visiting an artifact for the first time and the navigation method used to access it (for example, from the list of classes, or by executing a "references" search).

External validity

Arguably, the external validity of this study is limited since it used only two change tasks drawn from a single software development project. However, we tried to select study tasks that are representative of change tasks that might be given to a newcomer developer during his or her "assimilation"

[10] Intercoder reliability, or intercoder agreement, is the measure of the extent to which independent coders evaluate a characteristic of an artifact and reach the same conclusion.

phase. Both tasks involved adding new functionality to the system rather than fixing a bug, and this functionality had at least some visual component to it, to make it easier to initially explain the requirement to the newcomer. The tasks were limited in scope to a handful of different modules, but were not trivial in terms of their solution, which had to use methods from at least two different subsystems (for example, Java debugging and text editors in the case of the easy task).

Furthermore, the two tasks were very different from each other on a number of categories. First, the tasks came from two different subsystems of Eclipse, Team and Java, which are entirely independent of each other and whose development teams are even each located on a different continent. Second, the tasks differed in the amount of code necessary to implement the change, as well as its complexity and use of other classes and methods. Third, the help Hipikat provided in solving the tasks varied, with a top recommendation being extremely helpful in the easy task, while in the difficult task the participant had to examine several recommendations with differing levels of relevance to solving the task.

Lastly, Eclipse is quite typical of large open-source software projects in terms of artifacts that it produces in the course of the development, and development process that it follows. We believe that this similarity means that our approach would be applicable to other projects, such as Mozilla or Gnome, and that Hipikat would be able to recommend artifacts useful to project newcomers under those conditions as well.

4.2.9 Conclusion

Our case study showed that newcomers can use the information presented by Hipikat to achieve results comparable, or better, in quality and correctness to those of more experienced members of the team. In addition to validating this research claim, the study also allowed us to look at when newcomer developers query Hipikat, and how they evaluate Hipikat's recommendations.

The study participants used Hipikat in the early stages of their exploration, to give them an initial foothold in the system from which they can develop conjectures about the task and build their solution. Consequently, we observed the participants tended to stop their investigation of Hipikat's recommendations as soon as they felt that they have found a recommendation that was "good enough," and that would allow them to switch working with the code directly.

We also found difficulties for the newcomers to understand recommended artifacts in the context of the past system, and to take the knowledge forward and apply it to the current context. Partly, these difficulties were caused by the way recommended artifacts were presented, which made it difficult to understand their relationship with each other and the rest of the system. However, part of the problem lay in the change of usage context when code from a recommendation is being reused in a different location, something which Hipikat currently does not help in detecting.

4.3 A look at the quality of Hipikat's recommendations

To investigate the usefulness and accuracy of Hipikat for providing information relevant to a developer working on software modification task, we evaluated Hipikat's recommendations on a sample

of bug reports drawn from the Eclipse bug database.

We begin our description of the study by explaining how we selected the sample of bugs (Section 4.3.1) and the criteria by which we evaluated Hipikat's recommendations (Section 4.3.2). We then present a summary of the results, followed by a detailed description of three cases that represent the range of issues that we have identified in the results (Section 4.3.3).

4.3.1 Selecting the sample

Our study targeted modification tasks that were completed for Release 2.1 of Eclipse. In general, in open source software development, a bug report that is marked *fixed* corresponds to a single modification task performed by a developer. We thus initially defined the set of eligible reports as all reports from Eclipse's bug database that were marked "fixed" between June 27, 2002 (the day Eclipse 2.0 was released) and March 27, 2003 (the day of the 2.1 release). This is the same time period that we used in the Eclipse case study (Section 4.2), 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 bug reports to those modifications that a newcomer may have been assigned. Although there are no clear rules that can be used to automatically identify such modifications, a good approximation is to use the *severity* field of the bug report. A severity value of *minor* is defined in Eclipse.org's online help for Bugzilla [11] 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, if a bug's severity has been set to the value of "minor," it is virtually always a good indication that fixing the bug is an appropriate task for a project newcomer, and is for the purposes of this investigation the most practical method to select such tasks.

A total of 215 reports from the Eclipse bug database matched our criteria (bugs of severity "minor" which were resolved to "fixed" between June 27, 2002 and March 27, 2003). From this set, we randomly selected a sample of 20 bug reports 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 these reports made it into our sample was that the report was inappropriately marked as "fixed"—for example, if in reality the problem described in the report was determined not to be a real bug 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 report if we could not determine how it was fixed, which was usually when it was fixed as a side-effect of another (larger) modification task.

4.3.2 Evaluation criteria

The performance of information retrieval systems is often evaluated in terms of recall and precision [101]. 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.

[11] `https://bugs.eclipse.org/bugs/queryhelp.cgi#severity`

More formally, we call the *solution* of a modification task m the set of files $f_{sol}(m)$ that contain the implementation of the task. 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 in $\mathcal{H}(m)$:

$$f_r(m) = \{f_{sol}(b) | b \in \mathcal{H}(m) \wedge b \text{ is a modification task}\}$$

that is, the files that can be reached through the two-step investigation procedure of querying Hipikat first on the assigned task, and then on bug in the returned recommendation list that were marked as "fixed." This was the search strategy that was also used by the newcomer participants in our study (see Section 4.2.6).

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

$$precision(f_r(m)) = \frac{|correct(f_r(m))|}{|f_r(m)|}, \text{ where} \quad (4.1)$$

$$correct(f_r(m)) = f_r(m) \cap f_{sol}(m) \quad (4.2)$$

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:

$$recall(f_r(m)) = \frac{|correct(f_r(m))|}{|f_{sol}(m)|} \quad (4.3)$$

However, as we have seen in the Eclipse case study (Section 4.2), there is more value in Hipikat recommendations than just finding the location where the solution should be implemented. In particular, the usefulness of recommendations in the difficult task lay in their *examples of use* of the relevant APIs, rather than pointing to the class that contained the solution. Given the above measures of precision and recall, Hipikat recommendations for the difficult task in the Eclipse study would score poorly, although they were useful. We therefore propose to 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 to some extent a matter of individual judgement. (For example, it's not necessarily simply the number of method calls involved in the solution.) Instead, we will include the relevant constructs, and their presence in Hipikat recommendations, in the detailed description of results for individual bugs in the next section.

4.3.3 Results

The results of this investigation are summarized in Table 4.6. For each bug report in our sample, we list the precision and recall of recommended files and constructs. In each case we also include the rank of the first recommendation that contained the right files or construct, respectively.

Bug id	Files		First-useful recommendation			
	Precision	Recall	Files		Constructs	
			All bugs	*Completed*	*All bugs*	*Completed*
23719	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	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	0 (0/60)	0 (0/2)	N/A	N/A	None	
Avg.	0.11; 0.06	0.65; 1.00				

Table 4.6: Recall and precision of recommendations for a sample of 20 bug reports. The "Average" row gives mean and median of the sample. "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. Column "All bugs" indicates the absolute rank among recommended bug reports, while the "Completed" column gives the rank when taking into account only those recommendations that represented completed modification tasks that also had attached file revisions.

We will return to Table 4.6 in the following section (4.3.4), when we discuss study results in more detail. But first, in the remainder of this section, we describe in detail recommendations for three bugs from the sample. These bugs were selected as examples of a situation in which Hipikat made very good, moderately useful, or poor recommendations, respectively, and we use them to illustrate the kind of help that can be expected from Hipikat in practical situations.

An example of a very useful recommendation

Bug report 23719 points out an inconsistency in the 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 the setter will be named "setIsFoo," which causes interoperability problems with bean introspection applications.

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 field name is first stripped of pre-configured field prefixes and suffixes, like "f-" or "-_m".) The method is overloaded and exists with two signatures: in one case there is one parameter of type `IField`, in the other there are two, a `String` and a `boolean`. (The latter variant of the method existed with only the `String` parameter prior to this modification task.) The `proposeSetterName(IField)` method simply calls `proposeSetterName(String, boolean)` with the string argument set to `IField.getElementName()` and boolean set to `IField.getTypeSignature().equals(Signature.SIG_BOOLEAN)`.

The changes to the `NameProposer.proposeSetterName` method mean that the extra boolean parameter has to be added in its callers in other classes: `RenameFieldRefactoring`, `GetterSetterUtil` and `AddGetterSetterOperation`. Also, the method can now throw a `JavaModelException` and so a "throws" clause has to be added to `RenameFieldInputWizard`'s `createNewSetterName` method.

Lastly, an JUnit test was written (as class `NameProposerTest`) and added to the suite of tests for JDT UI plug-in (class `AutomatedSuite`).

Recommendations Hipikat turns out to be very helpful for solving this bug because it provides highly ranked recommendation that points out both the constructs and the locations of the solution. The top Hipikat recommendation for bug 23719 is bug report 6887. This report corresponds to the modification task that changed the name 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. The constructs used parallel the ones in the solution for bug 23719 in the `proposeSetterName` method that we described: if the field is a boolean, then the method adds the "is" prefix to the field name (with prefixes and suffixes stripped away first). The modification implements `proposeGetterMethod` as an overloaded method, which takes either a single "IField" parameter or a String and a boolean.

The `IField` variant of `proposeGetterName` calls the other one exactly the same way as the `proposeSetterName(IField)` calls `proposeSetterName(String, boolean)` in the solution to bug 23719.

These changes also necessitate adding the extra boolean parameter in what used to be `proposeGetterName(String)` to its callers in `RenameFieldRefactoring`, `GetterSetterUtil`, and `AddGetterSetterOperation`. Also, because `proposeGetterName` can now throw a `JavaModelException`, a "throws" clause had to be added to `RenameFieldInputWizard`'s `createNewGetterName` method.

Applying the recommendations As can be seen from the preceding descriptions, the solution to the top Hipikat recommendation for report 23719 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 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 23719 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 him- or herself, which given the complexity of the API for manipulating the model of Java programs in Eclipse, is very helpful.

Moderately useful recommendations

Bug report 6732 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. It is perhaps unavoidable that some operations will take longer to execute, but in this case Eclipse does not give any visual indication that the operation is in progress; the lack of response can even make it appear to the user that the application has locked up. The bug report concludes by suggesting that the widget follow accepted HCI practice of showing a busy cursor while node expansion is in progress.

Solution The solution, shown in Figure 4.10, is to perform the 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, of the four recommended completed modification tasks, three contain an implementation of the busy cursor in a different context (2937, task view filter, 15506, switching launch configs, and 9687, opening an editor). Each of the three useful modifications uses the `BusyIndicator.showWhile` construct, with the operation wrapped in a `Runnable`. The remaining completed modification task that was recommended but not useful (3790, busy cursor when opening type hierarchy) is linked, with low confidence, to three file revisions that seem to be

```
  protected void createChildren(final Widget widget) {
|   final Item[] tis = getChildren(widget);
    if (tis != null && tis.length > 0) {
      Object data = tis[0].getData();
      if (data != null)
        return; // children already there!
    }

>   BusyIndicator.showWhile(widget.getDisplay(), new Runnable() {
>     public void run() {
        // fix for PR 1FW89L7:
        // don't complain and remove all "dummies" ...
>       if (tis != null) {
          for (int i = 0; i < tis.length; i++) {
            tis[i].dispose();
          }
>       }
        Object d = widget.getData();
        if (d != null) {
          Object parentElement = d;
          Object[] children = getSortedChildren(parentElement);
          for (int i = 0; i < children.length; i++) {
            createTreeItem(widget, children[i], -1);
          }
        }
>     }
>   });
  }
```

Figure 4.10: Solution of bug 6732 (Diff-style view of changes implemented in AbstractTreeViewer revisions 1.11).

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 to realize *how* to implement this modification, although finding the exact location (AbstractTreeViewer) might be more of a challenge.

Unhelpful recommendations

Bug report 33182 is an example in which Hipikat did not provide any help, either to locate the file(s) involved, nor 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: if the information for one of the four applicable sections ("Supported platforms", OS, WS, and NL) is missing, the section will display a default value (e.g., "Supported platforms: all"). The solution is contained in two files: in class `DetailsForm` the setters for each of the four sections are extended to set the section's content to the default value if the assigned value is `null`, i.e., missing. The text for the default value is externalized in properties file `UpdateUIPluginResources.properties` for easy localization of the user interface. (See Sections 4.2.5 and 4.2.6.)

There are two related reasons for this 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 ten of them are from the area. To make things worse, 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 `DetailsForm`, 63 have no check-in comment at all, four had a bug number in the text, and one more 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.

4.3.4 Summary

Table 4.6 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, that is, it had a recall of 100%. In three more cases, Hipikat had a recall of 50% or more, recommending 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 6732 and 32067). Intuitively, this is going to be the case even with a perfect recommender if new functionality is being introduced into a module when it is already present in

other parts of the system (as in our example of a moderately successful Hipikat recommendation, bug 6732).

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 such modification in eleven out of sixteen cases for files and ten out of thirteen times for constructs—and always within the top four. This is important because we observed in the Eclipse 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 are counting on the developer to make reasonable choices when investigating Hipikat recommendations, which we believe is not an inappropriate assumption. Moreover, 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.

Likewise, although Hipikat 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 behaviour in the participants in the Eclipse study, and it has also been observed in other studies of programmer strategies of code reuse [97]. However, giving a smaller number of recommendations, and using different criteria to select an artifact for recommendation, remains an open area of research.

Chapter 5

Discussion

In this chapter, we describe the main issues that arose during the development and evaluation of Hipikat, the trade-offs involved, and our views on the potential impact of extended use of Hipikat on the process of software maintenance. Throughout the chapter, we compare our choices with existing alternatives, and note avenues for future research involving Hipikat.

5.1 Model

One comment that was common to participants in both the Eclipse and Avid studies (Sections 4.2 and 4.1, respectively) is that 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. It is already a fact of life in open-source software development. No serious project today comes without an online versioning system for the source code, an issue tracking system, archived mailing lists, and a web site; these same resources are available to every project on an open-source project hosting web site like SourceForge [1] or GNU Savannah [2]—and are so convenient that student teams in 4th year capstone software engineering courses sometimes use them for class projects. Commercial software development is going the same route: versioning control of source code and issue-tracking systems have been in common practice for some time. With the universal adoption of email, many organizations now have archives of developer communication for long-running projects. As the software industry is increasingly becoming involved in open-source projects (e.g., IBM, Sun, and Apple's participation in various high-profile projects is well-known), it can be expected that open-source techniques and tools will gain even wider acceptance.

In this section we will discuss some of the open issues regarding certain aspects of the Hipikat model, but will not question the basic assumption that those artifact repositories exist in the first place.

[1] `http://www.sourceforge.net`
[2] `http://savannah.gnu.org`

5.1.1 Unit of recommendation

Hipikat makes recommendation 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—e.g., Bugzilla for bug reports—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, however, it may be desirable to recommend only a portion of an artifact, e.g., if only a couple of comments in a long bug report were relevant to a query. Just what the appropriate portion of the artifact is could be highly dependent on the circumstances and has the potential drawback of a loss of context if the recommended fraction is too small, unless it is shown within the larger artifact. A way to accommodate this functionality within the existing Hipikat model would be to recommend entire artifacts and highlight the most relevant passages (similarly to the way Google highlights the search terms in cached pages). The highlights could be represented in a uniform way 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., [46, 100]). For a discussion of manual identification of such passages, see the section on collaborative filtering (Section 5.1.3).

At other times, the unit of recommendation should be a set of mutually related artifacts. A case in point is 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. Another example are threads of email messages or newsgroup articles. 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 node (cf. Figure 3.2) could either point to a single artifact, or (recursively) contain sets of recommendation nodes.

Lastly, in some cases the desired unit of recommendation does not map neatly onto the structure of artifacts in repositories. Sometimes the recommendation to a query could be a unit in the syntactic space of the programming language used in the project: a class or a method, for example (cf. [125] and [49] for examples of software development recommenders that operate at a class and method level of granularity). The content of such a recommendation could be obtained from a file version, especially if Hipikat could highlight the relevant range as discussed earlier in this section. However, the project memory model would have to be extended to allow entities at granularities other than artifacts in online repositories. It remains to be seen how far this metaphor could be stretched while still keeping it logically coherent to the user and maintaining the principle that each recommendation can be used as a starting point for another query.

5.1.2 Better time awareness

Although Hipikat records the time an artifact was created—and includes it in the information that is sent as part of a recommendation so the query results can be viewed sorted by creation date—it does not use this data in the recommendation heuristics. That is, it is not possible to ask Hipikat questions like "show me artifacts about feature X that were created since release N" (or even "since date D"). Such functionality could be trivially added by extending the search dialog feature (see Section 3.2.3) with special query conditions, although at the expense of increasing the complexity of the user interface.

Another time-related issue is that 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 a 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 have 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 [5]. 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 hasn't changed much over time, even old artifacts related to it will still be relevant. The information decay is not monotonous: 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.

5.1.3 Collaborative recommendation

As already discussed in Section 2.3, Hipikat's recommendations are 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. This is the approach we have implemented 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 at 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 should take into account the closeness of the current query and the query (or queries) for which a recommendation was given the thumbs-up or, a more typical pure collaborative recommending technique, the similarity in user profiles (e.g., interests or expertise) between the current users and the users who created the ratings. An even better solution would be to fully integrate collaborative and content-based criteria, such as introduced in Fab [107]. Such a combination could not only improve the quality of recommendations, but also avoid the

negative sides of the collaborative recommendation approach, such as handling users with unusual information needs compared to the rest of the population.

Using explicit ratings has a significant drawback, however: it requires the users to perform additional work which is not going to directly benefit them, and, as Grudin has noted [43], 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 empirical 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, does not only introduce room for error, but is actually disruptive to the user's main task—planning the software change.

Typical collaborative recommenders that use explicit ratings don't 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., Video Recommender [48]), music (e.g., Ringo [105]), or books (e.g., the reader reviews on Amazon.com, which include a rating). In Hipikat's target domain, we argue, that is not the case and an approach that uses *implicit* rating (i.e., inferred from users' actions) would be more appropriate and likely to succeed. While the best way to obtain these ratings for Hipikat should be empirically established, two basic approaches that have been used in other recommender domains are applicable in our case. Firstly, the action of viewing an artifact can be taken as a positive rating; this rating can be further quantified based on the length of time spent viewing it (e.g., as in GroupLens [59]) or how often the artifact is visited by the whole user community [121]. An alternative approach which may better capture the context in which the "rating" was made is to use the developers' history of navigation within the IDE and Hipikat to extract patterns in the temporal order of viewing of artifacts [19, 39]. By discovering these patterns, the tool can then recommend artifacts that were consistently accessed in similar contexts. The unresolved problem with navigation patterns remains how to identify the "dead ends" and distinguish them from useful paths. Especially newcomers exploring the code likely will not have the experience to recognize dead ends and by following such irrelevant paths will increase those artifacts' future rank. It seems that having an option in the user interface to explicitly rate recommendations (especially negative ratings) would be a useful feature in such cases.

5.1.4 Making sense of the group memory

In her research on how knowledge workers learn, Kidd noted the value of the *act* of making notes to the learning process, rather than the notes themselves [57]. In the domain of teamwork, Schmidt and Bannon emphasize that it is the active construction of meanings in a common information space which facilitates cooperative work, not simple provision of a shared database [104]. The same critique of the metaphor of organizational memory as a storage, with "remembering" equivalent to passive retrieving of data from the memory store rather than constructive process adapted to current situation, is echoed in Bannon and Kuutti [8].

On these grounds, an argument could be made against the use of a project memory recommender

such as Hipikat because it promotes passive retrieval of information. Ribak et al. further argue that a good CSCW system for knowledge sharing must nourish community building [93]. They apply this principle in ReachOut, a knowledge-sharing tool designed to encourage group interaction. ReachOut offers chat-like forums where users can ask questions from their colleagues. The discussion can be both synchronous and asynchronous, but the contents persist for a limited time only. As Ribak et al. say, "There is no point in creating more written, explicit knowledge; it is the real collaboration and the sharing of tacit knowledge that needs to be fostered. Providing access to previous discussions will imply the traditional 'search before you ask' " [93, p. 128].

There is obvious merit in this approach, and we do not think Hipikat could (or even should) entirely replace interaction with team members. Still, given the large amount of existing project artifacts, it would only make sense to use the available information to take some of the burden off one's colleagues, especially in an environment like open source software where there may be too many newcomers to give them appropriate mentoring support. In some cases, there may simply be no one who can help with the problem because that knowledge has been lost (especially if people involved have left the project). Also, the existing information provided by Hipikat could help a newcomer ask more informed and focused questions, something that can only be beneficial to all involved.

More importantly, Hipikat does not simply serve ready-made answers; its recommendations are more often just starting points for individual learning. Viewed this way, Hipikat actually promotes newcomers' actively making sense of past experiences and adapting them to current issues, which is an essential process in their transition into becoming experienced members of the development team.

5.2 Implementation

5.2.1 Presentation of query results

Hipikat currently presents its recommendations as a list consisting of a one-line short artifact description taken from the artifact's metadata (see Table 3.1), the reason for the recommendation, its relevance to the query, and confidence in the result (see Table 3.2). Displaying an artifact depends on its type: file revisions are displayed using the existing Eclipse facilities, either standalone in an editor or in a side-by-side comparison with its predecessor, highlighting the changes; displaying bug reports replicates in Eclipse the functionality of the existing Bugzilla front-end; and a URL is opened in an external web browser. Both the presentation of the recommendation list and displaying 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. It is the simplest to implement and works fine in most situations. However, it has significant drawbacks when the user's purpose is exploratory browsing of a collection. 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., [3]). This task can be made even easier when the clusters are automatically labelled with their most salient keywords (e.g., as in WebRat [99]). Document clustering (from cluster calculation to visual presentation) and document summarizing are active areas of research in information retrieval, and incorporating their advances in Hipikat would be a useful step to take. Moreover, it would be an interesting study in its own right how beneficial those features are for understanding unfamiliar software and finding code to reuse.

As we have already noted in Section 4.2.6, even once participants in the Eclipse 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 seeing a revision's changes highlighted and side-by-side with its predecessor, understanding how those changes work is 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 in a graphical form would go a long way to assist the user. Implementing such a feature, however, would require a significant amount of work, both to solve the technical challenges and to determine the most effective form for the presentation.

5.2.2 Scaling up

The main bottleneck in the heuristics that are currently used in Hipikat is using Latent Semantic Indexing for determining text similarity. LSI uses singular value decomposition (SVD), which is a costly operation in terms of both memory consumption and computation time. Even using workarounds to reduce the size of the matrix on which SVD is performed—as we currently do (see Section 3.2.2)—or to approximate its calculation using less expensive methods would likely not scale to larger document collections because it would eventually become too slow or too inaccurate. (LSI's precision-recall performance in general tends to degrade as document collections become very large and heterogeneous [20].) There is, however, ongoing research into scaling up LSI to handle such large selections. For example, Tang et al. distribute the document collection over a peer-to-peer network to reduce the cost of the search, combined with other techniques for reducing the cost of SVD [114]. They report several orders of magnitude better efficiency than LSI, while maintaining LSI's retrieval quality even for large and heterogeneous document collections. Incorporating such a technique into Hipikat should allow it to scale even to the projects of the scale and duration of Mozilla (six years of development and 260,000 reports in its Bugzilla database).

5.2.3 Check-in comment and activity-based matching

When the developer checking a file version into the repository adheres to the bug number convention, the version's check-in comment is in general a better indication that it fixes a particular bug. However, it is not perfect: some developers do not enter the bug id into their check-in comments, and others simply enter a blank comment for the majority of their check-ins. (This seems to be more of a problem in some Eclipse subteams than other, as we have seen in Section 4.3.3.) For this reason, we added the activity-based matching. It complements the check-in comment quite nicely,

simply because most developers work on only a few problems at a time, and mark them fixed in the bug-tracking system as they check the implementation into CVS. Most of the time, the two matchers identify the same links. When they diverge, we rank the check-in comment matcher's recommendation higher, because it is usually the correct one. For example, developers don't always mark the reports in the bug-tracking system as fixed immediately after they check the fix into the CVS, but do a sort of "mass cleanup" of their assignments when they find the time.

Sometimes, however, the check-in comment can be wrong, as illustrated in the post on the developers' mailing list shown in Figure 5.1. In this example, the activity matcher identified the

---

```
Date: Tue, 25 May 2004 12:13:35 -0400
From: Name removed
Subject: [platform-ui-dev] Mislabled commit
To: platform-ui-dev@eclipse.org
Reply-To: platform-ui-dev@eclipse.org

A fix for Bug 62707 was erroneously submitted with a comment that
suggested that the fix was really for 63849.  The affected files are:

org.eclipse.ui.workbench/Eclipse UI/.../ActivityViewerFilter.java, 1.1
org.eclipse.ui.workbench/Eclipse UI/.../messages.properties, 1.9
org.eclipse.ui.workbench/Eclipse UI/.../PerspContentProvider.java, 1.17
org.eclipse.ui.workbench/Eclipse UI/dots/SavePerspectiveDialog.java, 1.11
org.eclipse.ui.workbench/Eclipse UI/.../SelectPerspectiveDialog.java, 1.20

Sorry to get your hopes up if you were keen on 63849.  :)
```

Figure 5.1: Email posted to an Eclipse developers' list to correct a mislabeled code check-in.

correct related artifact (bug 62707). Unfortunately, the fact remains that it would be ranked lower than the log-matcher's (incorrect) recommendation, bug 63849. As we saw in the user studies in Chapter 4, this means that the correct recommendation will be less likely to be investigated, especially when it is contradicted by the check-in comment. The check-in comment probably carries even more weight for newcomers, who are forced to rely on it because they are not yet familiar with the code and cannot easily evaluate the purpose of a change just by looking at a set of file revisions.

5.3 Validation

5.3.1 The choice of methodology

One of the fundamental decisions that had to be taken in this research was whether to take a classic experimental approach that included a control group and was analyzed purely quantitatively, or to conduct a less-structured study that used a qualitative analysis approach. We opted for the latter in order to have a richer set of data and look at qualitative aspects of what the study participants did

and how they used Hipikat. This is reflected in the design and the kinds of questions we posed in the Eclipse user study (see Section 4.2). We believe this was the appropriate choice for this stage of the research, when we were interested in describing the situation and showing that our approach has some promise. Only now, following the results presented in this dissertation, can a true experiment be set up and the project memory approach evaluated more quantitatively [73].

A related validation design choice in the Eclipse study was not to use a true control group of newcomers who did not have access to Hipikat. Based on working with newcomers to Eclipse within our research lab and attending IBM Ottawa lab's Eclipse "training camp" for cooperative work students, we believe that newcomers placed into a situation that involved solving adding a feature to Eclipse in a limited time without access to either Hipikat or mentoring would fail to complete any significant part of the assigned task. To avoid this strawman case, we chose to compare newcomers using Hipikat with experts not using Hipikat. We believe that our finding that the newcomers' solutions were just as correct, or better, than the experts' should be an even more interesting result than what we could have got from a classic control group.

5.3.2 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 precision and recall of Hipikat recommendations (Section 4.3), 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 study (Section 4.2), the two change tasks used in the study, combined with the Eclipse.org project practices, meant that the most useful artifacts were 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 4.2.6), 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 their study task done. We do not believe that this detracts from the study's support of research claims; however, a closer examination of 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.

5.3.3 Measure of effectiveness

The primary measures we used in the Eclipse 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 release of the software. As we discussed in Section 4.2.8, while we tried to make them as objective as possible, there is always some room for a rater's subjective interpretation. In addition, the criteria could have been organized slightly differently, which would change at least somewhat the correctness scores.

It is important to recognize, however, that those 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. And yet, at least in this case, 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.

5.4 Impact of extended use of Hipikat

While one of our starting principles was that little or no change to the development process be required in order to use Hipikat, it would be interesting to see how extended use of Hipikat would affect the developers. 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 have used it during their "apprenticeship." For example, would the 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 then whether the 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. At that point, a feedback mechanism on the relevancy of Hipikat recommendations might be introduced into the development process. These recommendations could be evaluated together with the new code during code review, similarly to the process proposed by Terveen et al [116].

It also remains to be seen how useful Hipikat would prove for experts. Even experts at times have to work in an area of the system with which they have had little experience, that is, areas in which they are relative newcomers. Additionally, power users often come up with new ways to use a tool in ways that were never envisaged by the tool's designer. Anectodal evidence of Internet power users employing Google as a substitute for bookmarks in a web browser is a good example of this phenomenon. There is little incentive to spend time and effort maintaining one's bookmarks if a Google query can find a site just as quickly—and show other pages, including newly-created ones,

that match the same search terms and are potentially relevant to the current information needs. It is possible that expert software developers will similarly find their own idiosyncratic uses of Hipikat for applications unlike any described in this dissertation.

5.5 Hipikat's applicability

We conclude this chapter with a discussion of the range of situations in which we believe Hipikat to be particularly useful. However, it is important to first note the characteristics of a project's organization and culture that are the necessary preconditions for an environment in which Hipikat can be successfully introduced.

5.5.1 Environmental pre-conditions

In this research, we focused on a particular subset of virtual software development teams: open-source software (OSS). The reasons for this choice go beyond OSS projects' public repositories, which gave us ready access to information sources for the project memory and made it easy to set up a realistic environment for our validation studies (see Sections 4.2 and 4.3). Even more important was the culture of openness that permeates open-source software projects. Not only are almost all important project discussions conducted in public forums, such as developers' mailing lists; OSS developers are committed to re-posting and summarizing in such public forums even discussions that were—for whatever reasons—conducted in a different medium, for example face-to-face or in a personal email [45]. This committment extends to other development practices, such as the Mozilla project's public review of all software changes before they are committed to the code base in the repository or the obligatory reference to the bug being fixed in a check-in comment as the changes are entered into the source repository. At the same time, open-source software developers rarely keep thorough and up-to-date documentation; writing code is generally considered more fun (and challenging) and brings greater prestige within the project, and there are few other incentives that can be given to unpaid volunteers that form the bulk of contributors to OSS projects. 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 focused on projects that have relatively large developer teams (with ten 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; smaller projects rarely use all four information sources in our model, and if a team has only two or three developers—as we have seen in the example of unhelpful recommendations in Section 4.3.3—there is less need to fully discuss and review all modifications to the code. 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 by and the project grows and evolves, as illustrated by the examples below.

5.5.2 Hipikat's strengths

Given the above pre-conditions for a successful introduction of Hipikat into a project, we believe it would be the most useful in the following types of situations:

- refining existing functionality
- learning the API usage
- avoiding common errors
- recovering design rationale

Refining existing functionality

Eclipse bugs 6660 (the easy task in the Eclipse study, see Section 4.2.1) and 23719 (the "useful" case in the recommendation quality study, see Section 4.3.3) are both good examples of a situation where Hipikat provided information that was very useful for implementing the solution.

What both of these bugs had in common was that they were adding features that built on top of existing functionality that was added at a discrete point in time and a very distinct record in the project memory. Bug 6660 expanded the information displayed in a breakpoint hover pop-up, and extended the functionality that had previously worked only on conditional breakpoints to all breakpoints. Bug 23719 added boolean-specific naming of automatically-generated field "setter" methods; a functionality that could to a large extent be based on the existing generation of boolean-specific "getter" methods.

In each of these cases, the issue-tracking database contained a single report that corresponded to the existing functionality that was being refined—bugs 15739 and 6887, respectively. The report and the corresponding code implementing the solution were highly-rated Hipikat recommendations. This made it relatively easy to not only find the location for the modification, but also to see exactly how the base functionality was grafted onto the system when *it* was being implemented. Information that can be learned from the recommended code goes beyond just which method calls were used to implement the functionality; a newcomer could also see the accepted ways for error handling, architectural practices, and naming conventions. It should be noted, however, that in both cases the relevant parts of the system remained relatively stable since the functionality that was being refined had been implemented, which made it easier to learn from the recommendations and apply them to the new task.

Learning API usage

Eclipse bug 6732 (the "moderately useful" case in the recommendation quality study, see Section 4.3.3) is an example of a different kind of situation when Hipikat's recommendations can be helpful to a newcomer. Here, the newcomer does not have existing functionality to refine, but rather is implementing functionality that appears in other forms elsewhere in the system, but always follows the same pattern.

In this example, bug 6732 is about displaying a busy cursor while long-running operations are being executed. This functionality appears in a variety of other contexts in Eclipse. In many of these contexts, the busy cursor was added in response to a bug report so there is a distinct record in the project memory. Hipikat finds and recommends three such instances as related to bug 6732, and it is easy to see the relevant API and how it is to be used to "wrap" the existing operation.

While bug 6732 deals with the user interface, the same principle applies to patterns of actions at other levels of the code, such as logging and handling errors or internationalizing the application by externalizing messages to the user in a properties file. Taken even further, *any* kind of API usage patterns could be detectable and recommendable to the developer. Hipikat does not go quite as far, but Holmes and Murphy have implemented just such a system, which uses the existing code base as the source from which the API patterns are mined [49].

Avoiding common errors

Sometimes there are undocumented aspects of the API that cause problems to developers who are relatively unfamiliar with it. Bug 19761 is an example of this problem: it reports a NullPointer exception that turns out to be caused by making a method call before the object has been properly set up through a fairly complex, and non-obvious, sequence of steps. Hipikat also turns out to be helpful in this case because the bug report includes the Java stack trace with the exception. This trace appears in the Eclipse console window when the exception occurs during execution, and can be used as a Hipikat query, which will then recommend bug 19761, along with the explanation of the proper API calling sequence.

Recovering design rationale

Artifacts recommended by Hipikat are not only useful for learning the API, but also the rationale for certain functionality choices. A recommendation related to bug 20982 (the difficult task in the Eclipse study, see Section 4.2.1) provides a good example of this situation. Bug 20982 deals with the user interface for handling versioning operations on files in the workspace. One of the top recommendations, bug 10541, includes a lengthy discussion of several behaviour issues that arose when a related versioning functionality was added to another Eclipse module. (Six comments on the original proposal were added over a five-day period after the report was submitted, and another two over the next month and a half as the solution was being implemented). The same issues would have to be considered during the implementation of bug 20982, although its solution would be implemented somewhat differently because of internal module differences. Nevertheless, the discussion and the choices made in bug 10541 are relevant, especially for the sake of the consistency in behaviour in similar components across the entire system.

Chapter 6

Conclusion

New members of virtual software development teams—where members of the team are not collocated—face significant challenges coming up-to-speed on the project because it is difficult to get effective mentoring. Virtual teams generally do not have available light-weight communication channels that make possible informal everyday interactions among the team members. Consequently, newcomers have limited access to information that is typically obtained only through personal contact with one's more experienced colleagues.

The motivation for the work described in this dissertation is to help developers better perform modifications by providing them information from the project's history. The developers, especially in virtual teams, suffer because of lack of information necessary to build appropriate understanding of the system. A lot of information already exists, but it is not used to its full potential. Virtual software development teams typically generate large amounts of electronic artifacts in the course of their work. These artifacts are usually accessible through archives of the project's mailing lists, the source code versioning system, and the bug-tracking system. We believe that the collection of artifacts across these repositories implicitly forms a *project memory* for a software development.

The thesis of this dissertation is that newcomer software developers can use information from the project memory about past modifications completed on the project to help them effectively perform modification tasks to the system; that the project memory can be built largely automatically, requiring minimal adjustments in work practices of software developers; and that developers can be provided with tools to efficiently access this project memory to find useful information.

To validate the claims of this thesis, we have developed a general model of the project memory that incorporates the types of artifacts and relationships between them that are typically generated in the course of software development by virtual teams. This model could be instantiated for any project that follows such development practices; we provided a practical implementation, called Hipikat, that was instantiated for a large open-source software project, the Eclipse IDE. We then evaluated the effectiveness of this implementation in two empirical studies.

In the first study, eight volunteers used Hipikat to perform two modification tasks that added new functionality to Eclipse. The tasks were concrete and realistic: they were selected from enhancements implemented for a subsequent release of Eclipse. The eight volunteers had no experience developing for Eclipse, although they had previous software engineering experience working

on medium to large software systems. We compared their solutions to the solution adopted by the Eclipse team in the official release. We also compared the newcomers' performance with four members of the Eclipse team who were asked to perform the same two modifications without Hipikat. The results of our study show that the newcomers were able to use information from Hipikat recommendations in their solutions, and that their solutions were of comparable, and often better, quality than the experts', although the newcomers did take longer to implement the tasks. The study also allowed us to investigate when and how newcomers used Hipikat, and to identify the problems they faced evaluating and utilizing the recommendations.

In the second study, we evaluated the usefulness of Hipikat's recommendations on a sample of twenty modification tasks performed on Eclipse during the development of release 2.1 of the software. The tasks were appropriate to the kinds of tasks given to newcomers to the project during their ramp-up phase. The usefulness of the recommendations was evaluated by calculating their recall and precision values with respect to the files that contained the "official" solution for each of the sample tasks and the constructs comprising the solution. The results of the study show that in most of the cases examined in the study, 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. One example where Hipikat was entirely unhelpful was when the description of the modification was very terse, and—more importantly—members of the subteam responsible for the feature tended not to use the online artifact repositories to their fullest (e.g., they rarely entered the check-in comments). However, because the subteam in question was small and relatively isolated from the rest of the project at the time, arguably this was not the kind of situation where a project history approach would be appropriate or even needed.

6.1 Contribution

In addition to demonstrating the validity of the thesis statement, the research described in this dissertation makes the following five contributions to the field of software engineering.

First, we provide a general model for an automatically-generated project memory. As a consequence of the generality of the model, the techniques we used can, to a large extent, be re-applied to create a project memory for a different software development project, particularly if its development process is similar to the open source development process.

Second, we describe heuristics that we used to automatically infer linkages between artifacts in the project memory. These heuristics are a proof of the concept that automatic creation of the project memory is possible and a viable approach to creating group memories with minimal disruption to the existing group practices. The heuristics described in this dissertation are to a large extent based on my observations of work conventions and on informal communication with developers in a large open-source software project. These heuristics are likely applicable to other projects using such a development methodology and tools.

Third, we describe a specific implementation of a project memory for a large open-source software project, the Eclipse integrated development environment. We discuss the issues of scalability related to the project memory of this project. The experiences we gained and engineering issues we

identified will be useful to researchers who want to apply our approach to other projects. We also provide a usable tool to access the project memory. The tool is integrated into one of the most popular development environments today and could be used with little or no change to provide access to project memory instantiations for other projects.

Fourth, we provide an in-depth description of a methodology for empirical study of software evolution tasks under realistic conditions that allows replication and comparative evaluation. This methodology can be reused by researchers conducting similar studies of programmers performing evolution tasks and of software engineering tools used in such tasks.

Finally, we describe the design of a study for evaluating the performance of recommender tools in software engineering domain. Our design extends the existing measures used to evaluate recommender systems in software engineering context (e.g., [125, 124]) by taking into account not just whether the tool can identify entities that need to change in a software modification task (e.g., files), but also whether it can identify the constructs that are used within the change. This design could be useful for researchers wishing to evaluate similar software engineering tools.

6.2 Summary and future work

In conclusion, although Hipikat is still in a prototype stage, the idea of using project memories as an aid for newcomers shows promise. However, before Hipikat can be considered for adoption as a standard tool for open-source software projects, much work remains to be done. This works falls into four general categories.

First, heuristics used to generate recommendations should be further improved. This involves both better identification of linkages between artifacts in the project memory and better selection of artifacts relevant to a query. Techniques that seem particularly interesting include collaborative filtering and better modelling of the user's ongoing exploration activities, interests, and information needs (Section 5.1.3).

Second, these heuristics should be adapted to ensure their viability as the project memory grows. For example, the largest open-source software project, the Mozilla web browser, contains nearly four times more reports in its bug database than Eclipse (as of September 6, 2004, there were 258,217 reports in the Mozilla bug database, while Eclipse had 73,343). The current implementation of Hipikat most likely could not cope with that kind of size, even by the brute force approach of improving the hardware used. A more scalable approach, probably distributed in nature, should be investigated (Section 5.2.2).

Third, the Hipikat front-end should be further developed to make it easier to understand and utilize the recommendations. Displaying the recommendations in formats other than a flat list, or even topically clustering them, could make it easier for the user to see similarities between the recommendations and identify interesting ones (Section 5.2.1). Furthermore, visualizing the changes contained in recommended file revisions could make it easier to understand their functionality and interaction with the rest of the system. Lastly, combining Hipikat with a tool for investigating program concerns, like FEAT [96], would make it easier to explore the source code starting from information provided in Hipikat's recommendations.

Finally, once Hipikat is permanently adopted by a software development team, its impact on team's development practices would be a valuable research topic both on its own and as a driver for further development of the tool (Section 5.4). This is perhaps the most interesting aspect of the work that lies ahead because software development is a human activity and thus subject to all of the quirks and idiosyncrasies of human behavior. Weinberg was perhaps the first to point this out in his pioneering book [120]. He laid the groundwork for what followed as others began to realize how much common ground there is between the fields of software engineering and computer-supported cooperative. The research reported in this dissertation was inspired by this and has attempted to further the connection.

Bibliography

[1] Mark S. Ackerman and Thomas W. Malone. Answer Garden: A tool for growing organizational memory. In *Proceedings of the Conference on Office Automation Systems*, pages 31–39, 1990. 2, 21

[2] Kari Alho and Reijo Sulonen. Supporting virtual software projects on the Web. In *Proceedings of 7th International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE '98)—Workshop on Coordinating Distributed Sofware Development Projects*. IEEE Press, April 1998. 132

[3] Robert B. Allen, Pascal Obry, and Michael Littman. An interface for navigating clustered document sets returned by queries. In *Conference on Organizational Computing Systems (COOCS'93)*, pages 166–171, 1993. 110

[4] G. Antoniol, G. Canfora, A. De Lucia, and Merlo Merlo. Recovering code to documentation links in OO systems. In *Proceedings of the Sixth Working Conference on Reverse Engineering (WCRE'99)*, pages 136–144. IEEE Computer Society Press, 1999. 6

[5] Avi Arampatzis. *Adaptive and Temporally-Dependent Document Filtering*. PhD thesis, Katholieke Universiteit Nijmegen, Nijmegen, The Netherlands, 2001. 107

[6] David L. Atkins. Version sensitive editing: Change history as a programming tool. In Boris Magnusson, editor, *System Configuration Management*, volume 1439 of *Lecture Notes in Computer Science*, pages 146–157. Springer-Verlag, 1998. 27

[7] Thomas Ball and Stephen T. Eick. Software visualization in the large. *IEEE Computer*, 29:33–43, April 1996. 28

[8] Liam Bannon and Kari Kuutti. Shifting perspectives on organizational memory: From storage to active remembering. In *Proceedings of the 29th Hawaii International Conference on System Sciences (HICSS'96)*, volume 3, pages 156–167, 1996. 108

[9] Laszio A. Belady and M. M. Lehman. A model of large program development. *IBM Systems Journal*, 15(3):225–252, 1976. 1

[10] Lucy M. Berlin. Beyond program understanding: A look at programming expertise in industry. In *Empirical Studies of Programmers: Fifth Workshop*, pages 6–25, 1993. 1

[11] Lucy M. Berlin, Robin Jeffries, Vicki L. O'Day, Andreas Paepcke, and Cathleen Wharton. Where did you put it? issues in the design and use of a group memory. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 23–30. ACM Press, 1993. 2, 22

[12] Brian Berliner. CVS II: Parallelizing software development. In USENIX Association, editor, *Proceedings of the Winter 1990 USENIX Conference*, pages 341–352. USENIX, 22–26 January 1990. 50, 135

[13] Robert W. Bowdidge and William G. Griswold. Supporting the restructuring of data abstractions through manipulation of a program visualization. *ACM Transactions on Software Engineering and Methodology*, 7(2):109–157, April 1998. 72

[14] Ivan T. Bowman and Richard C. Holt. Reconstructing ownership architectures to help understand software systems. In *Proceedings of the Seventh International Workshop on Program Comprehension*, pages 28–37, Pittsburgh, PA, USA, 5–7 May 1999. IEEE Computer Society Press. 27

[15] Don Box, David Ehnebuske, Gopal Kakivaya, Andrew Layman, Noah Mendelsohn, Henrik Frysyk Nielsen, Satish Thatte, and Dave Winer. *Simple Object Access Protocol (SOAP) 1.1.* World Wide Web Consortium, 8 May 2000. 33

[16] K. C. Burgess Yakemovic and E. Jeffrey Conklin. Report on a development project use of an issue-based information system. In *Proceedings of ACM Conference on Computer-Supported Cooperative Work (CSCW'90)*, pages 105–118, 1990. 7

[17] John M. Carroll, Sherman R. Alpert, John Karat, Mary Van Deusen, and Mary Beth Rosson. Raison d'etre: Capturing design history and rationale in multimedia narratives. In *Proceedings of ACM SIGCHI Conference on Human Factors in Computing Systems (CHI'94)*, volume 1, pages 192–197, 1994. Color plates on page 478. 22

[18] John M. Carroll and Thomas P. Moran. Introduction to this special issue on design rationale. *Human-Computer Interaction*, 6(3–4):197–200, 1991. 7

[19] Matthew Chalmers, Kerry Rodden, and Dominique Brodbeck. The order of things: activity-centred information access. In *Proceedings of the 7th World Wide Web Conference*, volume 30 of *Computer Networks and ISDN Systems*, pages 359–367. Elsevier, 1 April 1998. 108

[20] C.-M. Chen, N. Stoffel, M. Post, C. Basu, D. Bassu, and C. Behrens. Telcordia LSI engine: Implementation and scalability issues. In *Proceedings of the Eleventh International Workshop on Research Issues in Data Engineering (RIDE' 01)*, pages 51–58. IEEE Press, April 2001. 110

[21] Peter P. Chen. The Entity-Relationship model—Toward a unified view of data. *ACM Transactions on Database Systems*, 1(1):9–36, March 1976. 31

[22] Yih-Farn Chen, Michael Y. Nishimoto, and C. V. Ramamoorthy. The C information abstraction system. *IEEE Transactions on Software Engineering*, 16(3):325–334, March 1990. 3

[23] Yih-Farn R. Chen, Glenn S. Fowler, Eleftherios Koutsofios, and Ryan S. Wallach. Ciao: A graphical navigator for software and document repositories. In *Proc. Int. Conf. Software Maintenance, ICSM*, pages 66–75. IEEE Computer Society, 1995. 64

[24] Fah-Chun Cheong. *Internet Agents: Spiders, Wanderers, Brokers, and Bots*. New Riders Publishing, Indianapolis, IN, USA, 1996. 38, 53

[25] Tzi-cker Chiueh, Wei Wu, and Lap-Chung Lam. Variorum: a multimedia-based program documentation system. In *Proceedings of the IEEE International Conference on Multimedia and Expo (ICME 2000)*, volume 1, pages 155–158, 2000. 24

[26] Jeff Conklin and Michael L. Begeman. gIBIS: A hypertext tool for exploratory policy discussion. *ACM Transactions on Office Information Systems*, 6(4):303–331, October 1988. 7

[27] D. Crocker. RFC 822: Standard for the format of ARPA Internet text messages, August 1982. 42

[28] Davor Čubranić and Gail C. Murphy. Automatic bug triage using text categorization. In *Procedings of the Sixteenth International Conference on Software Engineering and Knowledge Engineering (SEKE'04)*, pages 92–97. Knowledge Systems Institute, 2004. 28

[29] M. Cusumano and R. Selby. *Microsoft Secrets: How the World's Most Powerful Software Company Creates Technology, Shapes Markets, and Manages People*. The Free Press, 1995. 1

[30] The CVS project. The CVS client/server protocol specification. Included with the CVS source code distribution. 50

[31] Scott C. Deerwester, Susan T. Dumais, Thomas K. Landauer, George W. Furnas, and Richard A. Harshman. Indexing by latent semantic analysis. *Journal of the American Society of Information Science*, 41(6):391–407, 1990. 41

[32] P. Devanbu, Y.-F. Chen, E. Gansner, H. Müller, and J. Martin. CHIME: Customizable Hyperlink Insertion and Maintenance Engine for software engineering environments. In *Proceedings of the 21st International Conference on Software Engineering (ICSE'99)*, pages 473–482. ACM Press, 1999. 6

[33] Susan Dumais. Improving the retrieval of information from external sources. *Behaviour Research Methods, Instrument, and Computers*, 23(2):229–236, 1991. 40

[34] Susan T. Dumais. LSI meets TREC: A status report. In *Proceedings of The First Text REtrieval Conference (TREC1)*, pages 137–152, 1993. 41

[35] Michael E. Fagan. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 15(3):182–211, 1976. 23

[36] H. Fagrell. Newsmate: Providing timely knowledge to mobile and distributed news journalists. In *Beyond Expertise*, pages 257–274. 2003. 26

[37] G. Fischer, A. C. Lemke, R. McCall, and A. I Morch. Making argumentation serve design. *Human-Computer Interaction*, 6(3–4):393–419, 1991. 7

[38] J. C. Flanagan. The critical incident technique. *Psychological Bulletin*, 51:327–358, April 1954. 79

[39] Xiaobin Fu, Jay Budzik, and Kristian J. Hammond. Mining navigation history for recommendation. In *Proceedings of the 2000 International Conference on Intelligent User Interfaces*, pages 106–112, 2000. 108

[40] Pankaj K. Garg and Walt Scacchi. Ishys: Designing an intelligent software hypertext system. *IEEE Expert*, 4(3):52–63, Fall 1989. 6

[41] Daniel M. German. Mining CVS repositories, the softChange experience. In *Proceedings of the First International Workshop on Mining Software Repositories (MSR'04)*, pages 17–21, Edinburgh, UK, 25 May 2004. 42

[42] Rebecca E. Grinter. Using a configuration management tool to coordinate software development. In *Conference on Organizational Computing Systems*, pages 168–177, 1995. 26

[43] Jonathan Grudin. Groupware and social dynamics: eight challenges for developers. *Communications of the ACM*, 37(1):92–105, January 1994. 108

[44] Bjorn Gulla. Improved maintenance support by multi-version visualizations. In *Proceedings of the International Conference on Software Maintenance 1992*, pages 376–383. IEEE Computer Society Press, November 1992. 28

[45] Carl Gutwin, Reagan Penner, and Kevin Schneider. Group awareness in distributed software development. In *Proceedings of the 2004 ACM Conference on Computer supported cooperative work (CSCW'04)*, pages 72–81, Chicago, Illinois, USA, 2004. ACM Press. 114

[46] Marti A. Hearst. TextTiling: A quantitative approach to discourse segmentation. *Computational Linguistics*, 23(1):33–64, March 1997. 106

[47] James D. Herbsleb, Audris Mockus, Thomas A. Finholt, and Rebecca E. Grinter. An empirical study of global software development: Distance and speed. In *Proceedings of the 23rd International Conference on Software Engineering (ICSE'01)*, pages 81–90, Toronto, Canada, 12–19 May 2001. IEEE Computer Society. 2

[48] Will Hill, Larry Stead, Mark Rosenstein, and George Furnas. Recommending and evaluating choices in a virtual community of use. In *Proceedings of ACM SIGCHI Conference on Human Factors in Computing Systems (CHI'95)*, volume 1, pages 194–201, 1995. 108

[49] Reid T. Holmes and Gail C. Murphy. Using structural context to recommend source code examples. In *Proceedings of the International Conference on Software Engineering (ICSE'05)*, to appear. 106, 116

[50] E. Horowitz and R. C. Williamson. SODOS: A software documentation support environment—its use. *IEEE Transactions on Software Engineering*, SE-12(11):1076–1087, November 1986. 6

[51] M. Horton and R. Adams. RFC 1036: Standard for interchange of USENET messages, December 1987. 42

[52] George P. Huber. A theory of the effects of advanced information technologies on organizational design, intelligence, and decision making. *Academy of Management Review*, 15:47–71, January 1990. 20

[53] D. Hutchens and Victor Basili. System structure analysis: Clustering with data bindings. *IEEE Transactions on Software Engineering*, SE-11(8):749–757, 1985. 4

[54] E. Ide and G. Salton. Interactive search strategies and dynamic file organization. In G. Salton, editor, *The SMART Retrieval System—Experiments in Automatic Document Processing*, chapter 18. Prentice Hall, 1973. 40

[55] Severin Isenmann and Wolf D. Reuter. IBIS—A convincing concept... but a lousy instrument? In *Proceedings of the Conference on Designing Interactive Systems (DIS'97)*, pages 163–172, Amsterdam, The Netherlands, 1997. ACM Press. 7

[56] Fan Jiang, Ravi Jannan, Michael L. Littman, and Santosh Vempala. Efficient singular value decomposition via improved document sampling. Technical Report CS–1999–5, Duke University, 1999. 41

[57] Alison Kidd. The marks are on the knowledge worker. In *Proceedings of ACM SIGCHI Conference on Human Factors in Computing Systems (CHI'94)*, volume 1, pages 186–191, 1994. 108

[58] Donald E. Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, 1984. 5

[59] Joseph A. Konstan, Bradley N. Miller, David Maltz, Jonathan L. Herlocker, Lee R. Gordon, and John Riedl. GroupLens: Applying collaborative filtering to Usenet news. *Communications of the ACM*, 40(3):77–87, March 1997. 108

[60] Douglas Kramer. API documentation from source code comments: a case study of Javadoc. In *Proceedings of the 17th Annual International Conference on Computer Documentation (SIGDOC'99)*, pages 147–153, New Orleans, LA, USA. ACM Press. 5

[61] B. M. Lange and T. G. Moher. Some strategies of reuse in an object-oriented programming environment. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 69–73. ACM Press, 1989. 8

[62] Jintae Lee. SIBYL: A tool for managing group decision rationale. In *Proceedings of ACM Conference on Computer-Supported Cooperative Work (CSCW'90)*, pages 79–92, 1990. 7

[63] Jintae Lee and Kum-Yew Lai. What's in a design rationale? *Human-Computer Interaction*, 6(3–4):251–280, 1991. 7

[64] Timothy C. Lethbridge, Janice Singer, and Andrew Forward. How software engineers use documentation: The state of the practice. *IEEE Software*, 20(6):35–39, November/December 2003. 4, 5

[65] Henry Lieberman. Autonomous interface agents. In *Proceedings of ACM SIGCHI Conference on Human Factors in Computing Systems (CHI'97)*, volume 1, pages 67–74, 1997. 25

[66] Stefanie N. Lindstaedt and Kurt Schneider. Bridging the gap between face-to-face communication and long-term collaboration. In *GROUP'97: International Conference on Supporting Group Work*, pages 331–340, 1997. 24

[67] David C. Littman, Jeannine Pinto, Stanley Letovsky, and Elliot Soloway. Mental models and software maintenance. In *Empirical Studies of Programmers*, pages 80–98. 1986. 3

[68] Robert Lougher and Tom Rodden. Supporting long term collaboration in software maintenance. In *Conference on Organizational Computing Systems*, pages 228–238, 1993. 23

[69] Jonathan I. Maletic and Andrian Marcus. Supporting program comprehension using semantic and structural information. In *Proceedings of 23rd International Conference on Software Engineering (ICSE'01)*, pages 103–112, Toronto, Canada, 12–19 May 2001. IEEE Computer Society. 4

[70] Andrian Marcus and Jonathan I. Maletic. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *Proceedings of the 25th International Conference on Software Engineering (ICSE'03)*, pages 125–137, Portlan, OR, USA, May 3–10 2003. IEEE Computer Society. 7

[71] Vahid Mashayekhi, Chris Feuller, and John Riedl. CAIS: Collaborative asynchronous inspection of software. In David Wile, editor, *SIGSOFT'94: Proceedings of the Second ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE'94)*, pages 21–34. ACM Press, 1994. 23

[72] David W. McDonald and Mark S. Ackerman. Expertise Recommender: A flexible recommendation system and architecture. In *Proceedings of ACM Conference on Computer Supported Collaborative Work (CSCW 2000)*, pages 231–240, Philadelphia, PA, 2–6 December 2000. ACM Press. 28

[73] Joseph E. McGrath. Methodology matters: Doing research in the behavioral and social sciences. In Ronald M. Baecker, Jonathan Grudin, William A. S. Buxton, and Saul Greenberg, editors, *Readings in Human-Computer Interaction: Toward the Year 2000*, pages 152–169. Morgan Kaufman, San Francisco, CA, 1995. 112

[74] Diane McKerlie and Allan MacLean. Experience with QOC design rationale. In *Proceedings of ACM INTERCHI'93 Conference on Human Factors in Computing Systems – Adjunct Proceedings*, pages 213–214, 1993. 7

[75] Ettore Merlo, Ian McAdam, and Renato De Mori. Source code informal information analysis using connectionist models. In *Proceedings of IJCAI '93: The Thirteenth International Joint Conference on Artificial Intelligence*, volume 2, pages 1339–1344. 4

[76] Amin Michail. CodeWeb: Data mining library reuse patterns. In *Proceedings of the 23rd International Conference on Software Engeneering (ICSE'01)*, pages 827–828, Toronto, ON, Canada, May12–19 2001. IEEE Computer Society. 9

[77] Audris Mockus, Roy T. Fielding, and James Herbsleb. Two case studies of open source software development: Apache and Mozilla. *ACM Transactions on Software Engineering and Methodology*, 11(3):1–38, July 2002. 6, 39, 42

[78] Audris Mockus and James D. Herbsleb. Expertise Browser: A quantitative approach to identifying expertise. In *Proceedings of the 24th International Conference on Software Engineering (ICSE'02)*, pages 503–512. ACM Press, May 19–25 2002. 27

[79] Hausi A. Müller and Karl Klashinsky. Rigi—A system for programming-in-the-large. In *Proceedings of the 10th International Conference on Software Engineering (ICSE'88)*, pages 80–86. IEEE Computer Society Press, 1988. 64

[80] Gail C. Murphy, David Notkin, and Kevin Sullivan. Software reflexion models: Bridging the gap between source and high-level models. In *Proceedings of the Third ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE'95)*, pages 18–28. ACM Press, 1995. 64

[81] L. R. Neal. A system for example-based programming. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI'89)*, pages 63–68. ACM Press, 1989. 8

[82] C. E. O'Malley, S. W. Draper, and M. S. Riley. Constructive interaction: A method for studying human-computer-human interaction. In *Proceedings of IFIP INTERACT'84: Human-Computer Interaction*, pages 269–274, 1984. 60

[83] Open Source Initiative. The Open Source definition, 1997. http://www.opensource.org/osd.html. 132

[84] David Lorge Parnas. Software aging. In *Proceedings of the 16th International Conference on Software Engineering (ICSE'94)*, pages 279–287. IEEE Computer Society Press / ACM Press, 1994. 1

[85] J. M. Perpich, D. E. Perry, A. A. Porter, L. G. Votta, and M. W. Wade. Anywhere, anytime code inspections: Using the Web to remove inspection bottlenecks in large-scale software development. In *Proceedings of the 19th International Conference on Software Engineering (ICSE'97)*, pages 14–21. ACM Press, 1997. 23

[86] Peter Pirolli and John Anderson. The role of learning from examples in the acquisition of recursive programming skills. *Canadian Journal of Psychology*, 35:240–272, 1985. 8

[87] M. F. Porter. An algorithm for suffix stripping. *Program*, 14(3):130–137, July 1980. 40

[88] Colin Potts and Glenn Bruns. Recording the reasons for design decisions. In *Proceedings of 10th International Conference on Software Engineering (ICSE'88)*, pages 418–427, 1988. 7

[89] David F. Redmiles. Reducing the variability of programmers performance through explained examples. In *Proceedings of ACM INTERCHI'93 Conference on Human Factors in Computing Systems*, pages 67–73, 1993. 9

[90] Brent Reeves and Frank Shipman. Supporting communication between designers with artifact-centered evolving information spaces. In *Proceedings of ACM Conference on Computer-Supported Cooperative Work (CSCW'92)*, pages 394–401, 1992. 23

[91] Paul Resnick and Hal R. Varian. Recommender systems. *Communications of the ACM*, 40(3):56–58, March 1997. 24

[92] Bradley J. Rhodes and Thad Starner. Remembrance agent. In *The Proceedings of The First International Conference on The Practical Application Of Intelligent Agents and Multi Agent Technology (PAAM '96)*, pages 487–495, 1996. 25

[93] Amnon Ribak, Michal Jacovi, and Vladimir Soroka. 'ask before you search': peer support and community building with ReachOut. In *Proceedings of ACM Conference on Computer-Supported Cooperative Work (CSCW'02)*, pages 126–135, 2002. 109

[94] Gabriel Ripoche and Les Gasser. Scalable automatic extraction of process models for understanding F/OSS bug repair. In *Proceedings of the 16th International Conference on Software Engineering and Its Applications (ICSSEA-03)*, 2003. 28

[95] H. Rittel and M. Webber. Dilemmas in a general theory of planning. *Policy Sciences*, 4:155–169, 1973. 7

[96] Martin P. Robillard and Gail C. Murphy. Concern graphs: finding and describing concerns using structural program dependencies. In *Proceedings of the 24th International Conference on Software Engineering (ICSE'02)*, pages 406–416, New York, May 19–25 2002. ACM Press. 119

[97] Mary Beth Rosson and John M. Carroll. The reuse of uses in Smalltalk programming. *ACM Transactions on Computer-Human Interaction*, 3(3):219–253, 1996. 8, 92, 93, 104

[98] Mary Beth Rosson, John M. Carroll, and Christine Sweeney. A view matcher for reusing Smalltalk classes. In *Proceedings of ACM SIGCHI Conference on Human Factors in Computing Systems (CHI'91)*, pages 277–283, 1991. 8

[99] Vedran Sabol, Wolfgang Kienreich, Michael Granitzer, Jutta Becker, Klaus Tochtermann, and Keith Andrews. Applications of a lightweight, web-based retrieval, clustering, and visualisation framework. In *Proceedings of Conference on Practical Aspects of Knowledge Management (PAKM'02)*, volume 2569 of *Lecture Notes in Computer Science*, pages 359–368. Springer-Verlag, 2002. 110

[100] Gerard Salton, J. Allan, and Chris Buckley. Approaches to passage retrieval in full text information systems. In *Proceedings of the Sixteenth Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR'93)*, pages 49–58, 1993. 106

[101] Gerard Salton and Michael J. McGill. *Introduction to Modern Information Retrieval.* McGraw-Hill, New York, 1983. 40, 97

[102] Robert J. Sandusky, Les Gasser, and Gabriel Ripoche. Bug report networks: Varieties, strategies, and impacts in an OSS development community. In *Proceedings of the First International Workshop on Mining Software Repositories (MSR'04)*, pages 80–84, Edinburgh, UK, 25 May 2004. 28

[103] Daniel L. Schacter. Implicit memory: History and current status. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, 13(3):501–518, 1987. 2

[104] Kjeldt Schmidt and Liam Bannon. Taking CSCW seriously: Supporting articulation work. *Computer Supported Cooperative Work (CSCW): An International Journal*, 1:7–40, 1992. 108

[105] Upendra Shardanand and Patti Maes. Social information filtering: Algorithms for automating 'word of mouth'. In *Proceedings of ACM SIGCHI Conference on Human Factors in Computing Systems (CHI'95)*, volume 1, pages 210–217, 1995. 108

[106] Ben Shneiderman and John M. Carroll. Ecological studies of professional programmers: An overview. *Communications of the ACM*, 31(11):1256–1258, November 1988. 72

[107] Marko Balabanović and Yoav Shoham. Fab: content-based collaborative recommendation. *Communications of the ACM*, 40(3):66–72, March 1997. 25, 107

[108] Simon Buckingham Shum. Analyzing the usability of a design rationale notation. In Thomas P. Moran and John M. Carroll, editors, *Design Rationale: Concepts, Techniques, and Use*, pages 185–216. Lawrence Erlbaum, Hillsdale, NJ, 1996. 7

[109] Simon J. Buckingham Shum, Allan MacLean, Victoria M. E. Bellotti, and Nick V. Hammond. Graphical argumentation and design cognition. *Human-Computer Interaction*, 12(3):267–300, 1997. 7

[110] Susan Elliott Sim and Richard C. Holt. The ramp-up problem in software projects: A case study of how software immigrants naturalize. In *Proceedings of the 20th International Conference on Software Engineering (ICSE'98)*, pages 361–370, Kyoto, Japan, 19–25 April 1998. IEEE Computer Society Press / ACM Press. 1

[111] Janice Singer, Timoth Lethbridge, Norman Vinson, and Nicolas Anquetil. An examination of software engineering work practices. In *Proceedings of CASCON'97*, pages 209–223, Toronto, Canada, 10–13 October 1997. 3

[112] Elliot Soloway, Robin Lampert, Stan Letovsky, David Littman, and Jeannine Pinto. Designing documentation to compensate for delocalized plans. *Communications of the ACM*, 31(11):1259–1267, November 1988. 6

[113] Margaret-Anne Storey and Hausi Mueller. Manipulating and documenting software structures using SHriMP views. In *International Conference in Software Maintenance*, pages 275–285. IEEE Computer Society Press, 1995. 4

[114] Chunqiang Tang, Sandhya Dwarkadas, and Zhichen Xu. On scaling latent semantic indexing for large peer-to-peer systems. In *Proceedings of the 27th Annual International Conference on Research and Development in Information Retrieval (SIGIR'04)*, pages 112–121. ACM Press, 2004. 110

[115] Warren Teitelman and Larry Masinter. The Interlisp programming environment. *Computer*, 14. 3

[116] Loren G. Terveen, Peter G. Selfridge, and M. David Long. From "folklore" to "living design memory". In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 15–22. ACM Press, 1993. 21, 113

[117] Robert J. Walker, Gail C. Murphy, Bjorn Freeman-Benson, Darin Wright, Darin Swanson, and Jeremy Isaak. Visualizing dynamic software system information through high-level models. In *Proceedings of the 13th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA-98)*, pages 271–283. ACM Press, October 18–22 1998. 61

[118] Larry Wall and Randal L. Schwartz. *Programming Perl*. O'Reilly & Associates, 1990. 134

[119] James P. Walsh and Gerrardo Rivera Ungson. Organizational memory. *Academy of Management Review*, 16:57–91, January 1991. 20

[120] Gerald E. Weinberg. *The Psychology of Computer Programming*. Van Nostrand Reinhold, Toronto, 1971. 120

[121] Alan Wexelblat and Pattie Maes. Footprints: History-rich tools for information foraging. In *Proceedings of ACM SIGCHI Conference on Human Factors in Computing Systems (CHI'99)*, volume 1, pages 270–277, 1999. 108

[122] Steve Whittaker and Candace Sidner. Email overload: Exploring personal information management of email. In *Proceedings of ACM SIGCHI Conference on Human Factors in Computing Systems (CHI'96)*, volume 1, pages 276–283, 1996. 23

[123] Yunwen Ye and Gerhard Fischer. Information delivery in support of learning reusable software components on demand. In Yolanda Gil and David B. Leake, editors, *Proceedings of the 2002 International Conference on Intelligent User Interfaces (IUI-02)*, pages 159–166, New York, January 13–16 2002. ACM Press. 25

[124] Annie T. T. Ying, Gail C. Murphy, Raymond Ng, and Mark C. Chu-Carroll. Predicting source code changes by mining revision history. *IEEE Transactions on Software Engineering*, 30:574–586, September 2004. 27, 119

[125] Thomas Zimmermann, Peter Weisgerber, Stephan Diehl, and Andreas Zeller. Mining version histories to guide software changes. In *Proceedings of the 26th International Conference on Software Engineering (ICSE'04)*, pages 563–572. IEEE Computer Society, 2004. 27, 106, 119

Appendix A

Open-source software development

A.1 Introduction

In recent years a form of software development that was previously dismissed as too ad-hoc and chaotic for serious projects has suddenly taken the front stage. With products such as Apache, Linux, Perl, and others, open-source software has emerged as a viable alternative to traditional approaches to software development. With its globally distributed developer force and extremely rapid code evolution, open source is arguably the extreme in "virtual software projects" [2], and exemplifies many of the advantages and challenges of distributed software development.

According to its (trademarked) definition, open-source software (OSS) is software for which the source code is distributed or accessible via the Internet without charge and without limitations on modifications and future distribution by third parties [83]. While much of the early ARPANet and Unix software was distributed in this manner, more ambitious open-source projects such as Free Software Foundation's GNU began in the 1980's, and gained support of developers across the Internet. However, it wasn't until the 1990's that open-source software development truly gained momentum and became synonymous with highly distributed development characterized by frequent iterations, thanks to the wide availability of the source code and openness to contributions from the community.

Today, open-source software dominates the Internet infrastructure. For example, according to Netcraft's survey conducted in September 2004, over 67% of web servers run Apache server software, over three times as much as its next competitor, Microsoft's Internet Information Server,[1]. Programming languages that are developed and controlled by open-source communities—for example, Perl and Python—are among the few serious alternatives for interpreted languages used in "real-world" applications. Established computer companies such as IBM and Apple are including OSS into their products and actively support OSS projects.

The main reason for the success of open-source software is the growth of the Internet, which made collaboration between programmers feasible on a scale much larger than was possible before. With the global computer network in place, a huge pool of potential developers and testers became

[1] `http://news.netcraft.com/archives/2004/08/31/september_2004_web_server_survey.html`

available. Not surprisingly, this openness and fluidity also put unique demands on the development process. To cope with these issues, open-source software projects evolved their own methods and organization.

In this appendix, we look at the development tools and practices used by some of the major and most successful open-source projects to deal with the issue of coordination among their many contributors. Although each of the projects examined here developed some unique practices, there are also significant commonalities.

A.2 Current Practices

There are thousands of open-source projects that are currently under development. The ones we include in this section are examples that are notable for their influence, size, and success. They are in many ways the community leaders: a lot of tools and practices pioneered by these projects have been adopted by other open-source projects. Consequently, practices described in this section are arguably representative of open-source software in general.

A.2.1 Representative Open-Source Projects

Linux. Linux is arguably the best-known open-source project today. It is a Unix-type operating system kernel which aims for a complete implementation of the POSIX specification, with System V and BSD extensions. What started off in 1991 as a hobby project of Linus Torvalds, then a student at University of Helsinki, has evolved into a full-featured modern OS (consisting of more than 1.5 million lines of code) that today even comes installed on some models of computers from vendors like IBM and Sun, who have their own versions of UNIX that they sell. The development of Linux is still led by Torvalds, although there is no formal organization like that used in the other projects described here.

Apache web server The Apache web server originated in early 1995 as a set of patches to the then-popular HTTP server from NCSA (the name was also a pun, "A PAtCHy server"). These patches were collected by a group of volunteers from contributions from webmasters frustrated by NCSA's lack of further development and then released back to the web community. The patches were a big success, and soon the group moved on to a complete overhaul and redesign of the server. Apache 1.0 was released to the general public on December 1, 1995 and has dominated the web server market ever since.[2] The initial volunteers then formed the Apache Group, which in 1999 was transformed into a not-for-profit corporation, the Apache Software Foundation, which has been guiding the project ever since.

Mozilla web browser The Mozilla web browser is probably one of the most interesting experiments in open-source software development. It has been hugely influential in terms of project man-

[2]Netcraft's monthly surveys of web servers show Apache's share has been stable at around 67% since September 2003.

agement tools and practices used by large open-source software projects. In March 1998, Netscape released the source code for the next version of its Communicator web browser under an open-source style license, an unprecedented step at the time. Netscape created an independent entity, the Mozilla Organization or "Mozilla.org", to coordinate the developers' effort and act as a central point of contact for those interested in participating in the project. Mozilla.org provides support for the developers, including a web site and mailing lists, and publishes the Mozilla browser as its integrated version of the project's effort. Netscape remained actively involved in the project for several years, providing funding and personnel for mozilla.org and doing much of the development work. In 2003, Netscape (by then a division of AOL-Time Warner) established a not-for-profit independent corporation, the Mozilla Foundation, to provide a legal and organizational framework to the project.

Perl Perl is a general-purpose programming language, invented in 1987 by Larry Wall as a quick hack to simplify generating reports from systems logs [118]. It has since become the language of choice for small-to-medium projects, especially in the areas of World Wide Web development, system administration, and text processing. Today, the infrastructure for communication and coordination in support of the Perl community is provided by the Perl Foundation, a non-profit organization.

Eclipse Eclipse is an extensible integrated development environment (IDE) that began as an internal project at IBM and was released under an open source license in 2001. It is rapidly increasing in popularity among IDEs for Java development in particular, and is also used as a platform underlying commercial products, such as IBM's WebSphere family of development tools. When IBM turned Eclipse into an open-source project, it also created the Eclipse Consortium to guide and oversee its development. In early 2004, the Consortium was superseded by the Eclipse Foundation, an independent not-for-profit corporation.

A.2.2 Communication and Coordination

One of the most important characteristics of distributed software development is that developers can no longer rely on face-to-face meetings, but instead have to make use of technology to allow them to communicate over distance. Open-source projects rely on remarkably simple computer-based communication technologies, such as mailing lists and newsgroups, for almost all communication activities. Furthermore, coordination among the participating developers regarding the direction of the project, code review, and for some projects even bug reporting and code contributions, is all conducted through such media.

There are two main reasons for choosing such a relatively low-tech approach. First, the extremely distributed nature of even a core group of developers of OSS projects—for example, Apache's 20 core developers are located in five different countries across three continents—all but precludes the usage of synchronous communication. Second, open-source development is extremely fluid, where structure is minimal and developers' contributions vary with time depending on their interest and other commitments. In such circumstances, it would be very difficult to impose a prescrip-

tive coordination technology, such as workflow systems. Thus, most projects prefer to continue using email, or email-like mechanisms, even after they have grown beyond the initial small group of developers because it lets humans resolve any unexpected situations that may arise.

Many OSS projects maintain web-based archives of their mailing lists, which ensure the future availability of information, and the archived discussions represent a form of a project's "self-documentation."

A.2.3 Version and Configuration Management

By the very definition of open-source software, such projects have to have a code repository on the Internet, where developers and other interested parties can access it for download. In the past, this was done mainly through anonymous FTP access to the recent versions (often there are at least two: the current "leading-edge development" and "stable" versions). Also, typically the changes from the previous version are available as *patch files*, available to those who want to minimize the amount of necessary downloading while keeping current with the ongoing development.

As projects grew in code and team size, they began to use configuration management tools to control the code repository and ease the burden of version control and merging in individual submissions. Today, CVS [12] (itself an open-source project) is almost universally the tool used for this purpose. Linux is a significant holdout in this department, because Linus Torvalds decided to use a commercial tool, BitKeeper,[3] to maintain his, "official," version of the Linux development tree.[4]

The organization of configuration management in open-source projects stems from their internal developers' "hierarchy". Typically, there are two tiers of developers participating in the effort: a core group that is relatively small (for example, around 20 people in the Apache project), and a much larger pool of contributors. The core developers are actively involved (often on a daily basis) in the development of the product, and some minimum amount of commitment in terms of time and effort is usually implicitly expected. Contributors might submit an occasional bug fix or feature enhancement, when they have time, interest, or ideas. The core developers are then expected to receive and review those contributions, and integrate the accepted ones into the code base. Over time, contributors who have distinguished themselves by the quality and frequency of their work may be invited to join the core group and gain more responsibility in the project. In other words, open-source projects operate as *meritocracies*.

Despite similarities across projects, there are individual project differences evident. In Linux, the final authority and say on what goes into the kernel rests with Linus Torvalds, although most responsibility over subsystems has been delegated to the so-called "module maintainers" (who have often also written a major portion of the code they oversee). In Apache, the core developers form

[3] `http://www.bitkeeper.com`

[4] Linux is somewhat unusual among large open-source projects because it still uses email to share modifications to the source code. These are then manually integrated into code repositories. Anybody is free to maintain their own version of the kernel, kept in a separate repository and built from a customized selection of posted modifications. In practical terms, these function roughly like development branches in CVS repositories. The official releases are always created from Torvalds's "main" development tree.

a group which maintains control over the whole project without further breakdown of the hierarchy by subsystem—essentially, all the members of the group have equal vote if an issue becomes contentious. Mozilla's and Eclipse's development is divided into subsystems. Development in each subsystem is overseen by "module owners." "Ownership" is a relative term, however; in theory, the module owners are only caretakers and can be replaced if the developers' community is dissatisfied with their work.

This sort of developer organization is reflected in the setup of the code repositories: in general the CVS source code repository allows read-only access to anyone on the Internet, while only the core developers have the permissions to directly modify the tree. Those programmers (the "committers") are responsible for evaluating and approving changes submitted by the community and integrating them into the tree. Mozilla introduces another layer into the hierarchy: all code that is to be checked into the repository by a committer needs to be reviewed by the relevant module owner (or one of his designated assistants) and further "super-reviewed" for integration with the rest of the system by a member of a small group of developers with long experience and broad expertise in the project.

A.2.4 Bug and issue tracking

A publicly accessible bug and issue tracking tool is used by nearly every significant open source project, with the notable exception of Linux.[5] In general, anyone can enter bug reports and enhancement requests into the system. (The distinction between the bugs and enhancements is sometimes not a sharp one, and terms "bug" and "bug report" are usually used to include enhancement requests as well.)

Most bug-tracking systems allow posting of additional comments in bug reports. In such systems, each bug report becomes, in effect, a tiny public mailing list focused solely on that issue. Some bugs are resolved rapidly and without discussion (e.g., "not a bug"). In others, the discussion can involve many participants, last for weeks, and include dozens of messages. In some projects (such as Mozilla), the code implementing a bug fix (a *patch*) is also included in the bug's discussion, where it can be reviewed for correctness and prompt additional discussion of APIs or coding alternatives. Because there is no restriction on who can add comments in a bug discussion, posting patches is a way for contributing source code to the project from programmers without commit privileges to the code repository.

The issue tracking tool is also used for project management. Each report has a number of attributes, such as "state" (starting from "new" and ending in "resolved"), severity (e.g., "minor" or "critical"), and the developer nominally in charge of the bug. The bug database can be queried on their description and attribute values, for example to find all bugs related to user interface, or to check which ones have remained unsolved for a long time.

Bugzilla, the issue-tracking tool that originated in the Mozilla project, is now used by all projects we described in this review, and is generally considered the dominant issue-tracking system for

[5] An experimental bug tracking system has been set up in 2003 for the current release of Linux, although the project's mailing list still remains the main forum for bug reporting and management.

open-source software. Other major issue-tracking tools—for example, SourceForge, used by the eponymous OSS project-hosting site [6]—have similar feature sets and are used in similar ways in the development process.

[6] http://www.sourceforge.net

Appendix B

Sample protocol between Hipikat client and server

This appendix presents the "gory details" of all exchanges between Hipikat client and server during the example session used in Section 1.2.2. The purpose is to document the details of the protocol at a level that would have made Chapter 3 a very tedious read indeed.

B.1 Acquiring user id

As described in Section 3.2.1, queries for Hipikat recommendations contain a unique id to anonymously identify the user. Each client locally stores this id once it is obtained. For the purposes of this example, we will assume that our fictional developer has a freshly installed Hipikat client, which has not yet been given a user id by the server. The first time the client needs to make a query to the server, it will first issue a request for a new user id. The request is essentially a SOAP RPC call to the remote method `createUserId`, and this is the message that is sent to the server:

```
<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope
    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
    xmlns:xsd="http://www.w3.org/1999/XMLSchema">
<SOAP-ENV:Body>
  <ns1:createUserId xmlns:ns1="urn:RecommendationFetcher"
    SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
  </ns1:createUserId>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

The server will respond with a new unique user id, which the client will use in future requests. In this case, the user id that is returned is "7so".[1]

[1]The user id is currently generated by increasing an integer counting the number of requests so far and converting it to base 36 so it's a mixture of letters and digits. The protocol does not depend on any particular

```
<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope
    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
    xmlns:xsd="http://www.w3.org/1999/XMLSchema">
<SOAP-ENV:Body>
  <ns1:createUserIdResponse xmlns:ns1="urn:RecommendationFetcher"
    SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
    <return xsi:type="xsd:string">7so</return>
  </ns1:createUserIdResponse>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

B.2 Making the first query

The first user's query to Hipikat in our example is on the assigned request, bug 6660 in the issue database. The query is a SOAP RPC call to a remote method `getRecommendation`, with parameter `userId` set to "7so," as assigned in the previous step. Parameter `artifactKey` is set to the key representing bug 6660, "bugzilla:6660" (c.f. Table 3.1). The value of the optional third parameter, `contextKey`, is set to "null," to indicate that it is not being used. (This argument is reserved for future extension to the protocol for describing the context of the query.)

```
<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope
    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
    xmlns:xsd="http://www.w3.org/1999/XMLSchema">
<SOAP-ENV:Body>
  <ns1:getRecommendation xmlns:ns1="urn:RecommendationFetcher"
    SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
     <userId xsi:type="xsd:string">7so</userId>
     <artifactKey xsi:type="xsd:string">bugzilla:6660</artifactKey>
     <contentKey xsi:type="xsd:string" xsi:null="true"/>
  </ns1:getRecommendation>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

The server responds with a recommendation list visible in Figure 1.7. Note that the result returned in the SOAP response in this case is a string encoding an XML structure, and correspondigly the angle brackets inside the response have been replaced with encodings "<" and ">". When the client receives the response, it will extract the contents of the `return` element, convert them back to regular XML to get the `RecommendationList`, and then parse it to obtain recommendations that can be presented to the user.

form of the unique id, as long as it is a string of characters.

```
<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope
    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
    xmlns:xsd="http://www.w3.org/1999/XMLSchema">
<SOAP-ENV:Body>
  <ns1:createUserIdResponse xmlns:ns1="urn:RecommendationFetcher"
    SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
    <return xsi:type="xsd:string">&lt;RecommendationList&gt;
  &lt;Recommendation&gt;
    &lt;key&gt;bugzilla:15739&lt;/key&gt;
    &lt;name&gt;Bug 15739 - Show breakpoint condition in hover help&lt;/name&gt;
    &lt;Created&gt;2002-05-10 13:59:00&lt;/Created&gt;
    &lt;lastModified&gt;2002-05-16 16:22:50&lt;/lastModified&gt;
    &lt;reason&gt;Text similarity&lt;/reason&gt;
    &lt;confidence&gt;0.498139442569073&lt;/confidence&gt;
  &lt;/Recommendation&gt;
  &lt;Recommendation&gt;
    &lt;key&gt;bugzilla:8679&lt;/key&gt;
    &lt;name&gt;Bug 8679 - Hit count ignored on breakpoints&lt;/name&gt;
    &lt;Created&gt;2002-01-29 15:57:00&lt;/Created&gt;
    &lt;lastModified&gt;2002-01-29 16:58:22&lt;/lastModified&gt;
    &lt;reason&gt;Text similarity&lt;/reason&gt;
    &lt;confidence&gt;0.370469849983177&lt;/confidence&gt;
  &lt;/Recommendation&gt;
  ...
  &lt;Recommendation&gt;
    &lt;Key&gt;news:www.eclipse.org/9v4pg0$f4j$1@rogue.oti.com&lt;/Key&gt;
    &lt;Name&gt;'JDT conditional breakpoints',
       posted by ynh@brf.dk on 2001-12-11 03:08:48.0&lt;/Name&gt;
    &lt;Reason userRecommended="false"&gt;Text similarity&lt;/Reason&gt;
    &lt;Created&gt;2001-12-11 11:08:48&lt;/Created&gt;
    &lt;lastModified/&gt;
    &lt;Confidence&gt;0.399561491312874&lt;/Confidence&gt;
  &lt;/Recommendation&gt;
  ...
&lt;/RecommendationList&gt;</return>
  </ns1:getRecommendationResponse>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

B.3 Making the second query

The user next opens bug report 15739, and upon deciding that it looks relevant to the task, issues a Hipikat query on it:

```
<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envolope  ...
<SOAP-ENV:Body>
  <ns1:getRecommendation  ...
     <userId xsi:type="xsd:string">7so</userId>
     <artifactKey xsi:type="xsd:string">bugzilla:15739</artifactKey>
     <contentKey xsi:type="xsd:string" xsi:null="true"/>
  </ns1:getRecommendation>
```

```
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Server's response is as before, except in this case log- and activity-matcher give a number of recommendations that appear at the top of the list. (SOAP headers are omitted below and only the actual `RecommendationList` shown.)

```
<RecommendationList>
  <Recommendation>
  <Key>cvs:org.eclipse.jdt.debug/model/org/eclipse/jdt/internal/)
      debug/core/breakpoints/JDIDebugBreakpointMessages.properties:1.4</Key>
    <Name>org.eclipse.jdt.debug/model/org/eclipse/jdt/internal/)
      debug/core/breakpoints/JDIDebugBreakpointMessages.properties:1.4</Name>
    <Reason>Bug ID in revision log</Reason>
    <Created>2002-05-15 18:58:50</Created>
    <lastModified/>
    <Confidence>High</Confidence>
    </Recommendation>
  <Recommendation>
    <Key>cvs:org.eclipse.jdt.debug/model/org/eclipse/jdt/internal/)
      debug/core/breakpoints/JavaBreakpoint.java:1.45</Key>
    <Name>org.eclipse.jdt.debug/model/org/eclipse/jdt/internal/)
      debug/core/breakpoints/JavaBreakpoint.java:1.45</Name>
    <Reason>Bug ID in revision log</Reason>
    <Created>2002-05-15 18:59:32</Created>
    <lastModified/>
    <Confidence>High</Confidence>
    </Recommendation>
  <Recommendation>
    <Key>cvs:org.eclipse.jdt.debug/model/org/eclipse/jdt/internal/)
      debug/core/breakpoints/JavaLineBreakpoint.java:1.31</Key>
    <Name>org.eclipse.jdt.debug/model/org/eclipse/jdt/internal/)
      debug/core/breakpoints/JavaLineBreakpoint.java:1.31</Name>
    <Reason>Bug ID in revision log</Reason>
    <Created>2002-05-15 18:59:21</Created>
    <lastModified/>
    <Confidence>High</Confidence>
    </Recommendation>
  ...
<RecommendationList>
```

B.4 Giving a thumbs-up to a recommendation

Because bug 15739 turned out to be so useful in solving this task, the developer goes back to the recommendation list of bug 6660 and gives the recommendation on bug 15739 a "thumbs up." Hipikat client issues a SOAP RPC to the remote method `rateRecommendation`, and this is the message that is sent to the server:

```
<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope ...
<SOAP-ENV:Body>
  <ns1:rateRecommendation ...
     <userId xsi:type="xsd:string">7so</userId>
```

```
      <artifactKey xsi:type="xsd:string">bugzilla:15739</artifactKey>
      <queryContext xsi:type="xsd:string">bugzilla:6660</queryContext>
      <rating xsi:type="xsd:string">1</rating>
  </ns1:rateRecommendation>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Parameter `userId` is used as before. Parameters `artifactKey` and `queryContext` identify the artifact which is given a thumbs up and the artifact that the query was on, respectively. In this example, the thumbs-up is being given to bug 15739 that was recommended in response to the query on bug 6660.

The server's response is a just an empty message since no value needs to be returned:

```
<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope ...
<SOAP-ENV:Body>
  <ns1:rateRecommendationResponse  ...
  </ns1:rateRecommendationResponse>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

B.5 Hipikat search

This section illustrates Hipikat functionality that is not used in the example we followed so far, although it is mentioned in Section 3.2.3: searching the artifact database using search terms given by the user.

The search functionality is accessed through a "Hipikat search" pane in the Eclipse search dialog. The user types in the search terms into the dialog and clicks on the "Search" button. Hipikat client makes a recommendation request to the server that is just a special case of querying on an artifact. The only difference is that the artifact key used is `search:expression%3D`*URL-encoded search terms separated by URL-encoded&'s*. For example, user's search on terms "breakpoint hover" would be sent in the following SOAP request:

```
<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope ...
<SOAP-ENV:Body>
  <ns1:getRecommendation   ...
     <userId xsi:type="xsd:string">7so</userId>
     <artifactKey xsi:type="xsd:string">search:expression%3D
       breakpoint%2Bhover%</artifactKey>
     <contentKey xsi:type="xsd:string" xsi:null="true"/>
  </ns1:getRecommendation>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

The server's response is a `RecommendationList`, as described earlier.

Appendix D

Participant questionnaire used in the Eclipse study

In our study of usage of tools and various information sources used in software development, knowing a developer's previous experience with Eclipse and other software development technologies is important if we are to draw useful conclusions from our observations. For this reason, please fill out the following questionnaire. It will be used only for the purposes of this research, and no publications using this data will contain information that could personally identify you.

1. How much experience have you had with Java?

   ☐ heard of it

   ☐ used it in some classes

   ☐ comfortable programming in it, familiar with the standard libraries

   ☐ can virtually think in it

   ☐ other (please describe)

1b. What other programming languages do you know well (that is, would have answered "comfortable" or above on the previous scale)?

2. How much experience have you had with Eclipse?

   ☐ heard of it

   ☐ used it a few times

   ☐ use it as my main development environment (for how long)?

   ☐ tried implementing simple plug-ins from tutorials (which ones?)

   ☐ programmed plug-ins for it independently and more extensively (please describe below)

   ☐ other (please describe)

3. Have you ever worked on a large project (in terms of either or both length and number of people involved)?

- ☐ never
- ☐ yes: please describe below the nature and size of project and the extent of your involvement with it

4. Have you ever worked on an open-source project?

   - ☐ never heard of "open source"
   - ☐ have used open-source software, haven't worked on one
   - ☐ released my own code under open-source
   - ☐ got involved in somebody else's project (please describe the nature and size of the project and the extent of your involvement with it)

5. Have you ever worked with a bug report/problem management system (e.g., Bugzilla)?

   - ☐ never heard of Bugzilla or bug reporting
   - ☐ have heard of it, never used it
   - ☐ have written some bug/problem reports myself (using which bug report system?)
   - ☐ used it as a developer (which bug report system? which project? how extensively?)

6. Have you ever worked with version management system (e.g., CVS, RCS)?

   - ☐ what is "version management"?
   - ☐ have heard of it, never used it
   - ☐ used it in my own project, but didn't have to share it with others (which system?)
   - ☐ used it in a team project (which system?)
   - ☐ other (please describe):

7. How much experience have you had with "design patterns"?

   - ☐ never heard of them
   - ☐ heard of them, mentioned in class
   - ☐ know of a few
   - ☐ could recognize them in code
   - ☐ use them regularly
   - ☐ other (please describe):

8. Do you use "alpha" software? Please describe which products and your reasons for using them.

8b. What do you do when such software crashes? (For example, go into the code to fix it, report the problem to its developers, restart it and try again, etc.)

9. How do you usually search for documents on the Web?

   ☐ type in a few words into a search engine (which one?) and see what happens

   ☐ try refining searches (please describe how)

9b. Do you use multiple information sources and search engines? Please describe which ones.

10. How do you typically go about solving a problem or designing a new feature?

11. How do you usually search for information relevant when doing software development? Which information sources do you usually use? (For each source, please describe how often you use it compared to others, in which circumstances you tend to use it, and why.) Also, do you use any special strategies to remember that information? (For example, keep a notebook or an online journal, bookmarks in a web browser and/or Eclipse.)

Appendix E

Participant instructions used in the Eclipse study

The purpose of this study is to help evaluate whether and how a particular new software tool, called Hipikat, can aid in the performance of software change tasks. The tool directs a software developer who is unfamiliar with a system to information relevant about the task that he is performing on the system. The information suggested is derived from the recorded history of prior completed changes and enhancements to the system. The proposed study has two objectives. The first objective is to determine whether the tool suggests useful information to developers. The second objective is to gather realistic information to refine and improve the algorithms used to suggest relevant material.

E.1 Change Plan

We ask you to make a change to the code of the Eclipse platform. The functionality needed is described in bug *id* in Bugzilla.[1] You can view the bug report in Eclipse by opening the **Bugzilla** pane in the **Search** dialog and typing the bug number into the "Bug id or summary terms" field. Please read the bug now and understand what it is asking. Once you have done this, please notify the investigator.

E.1.1 Task

1. Now try to prepare a change plan. You can write the plan into a file or on a piece of paper. Try to make the plan as detailed as possible: it should contain the list of files and methods to be modified, and a summary of the modifications. For example:

   File1.java
   Add a field and the corresponding getters and setters...

[1] The instructions were identical for each of the two change tasks, regardless of the order in which they were performed. Only the id of the bug report used in the task was changed. (Bug 6660 for the easy task and bug 20982 for the hard task.)

File2.java

...

etc.

2. To make the change plan, try using Hipikat as much as you can. If you want to, you can use any other Eclipse tools we described earlier (including the debugger), or the documentation available on Eclipse.org,[2] articles,[3] online help,[4] or search[5] the newsgroup and mailing list archives. If you think you are stuck or are unsure of what to do next, try making a Hipikat query!

3. You have 1 hour to complete the plan. The investigator will ask you a couple of brief questions about your progress at the half-way point.

4. Your plan must be as complete as possible.

5. If you are ready to go, please notify the investigator.

E.2 Performing the change

Now start working on implementing the change you have planned.

1. Try to implement the functionality requested in bug *id*.

2. Feel free to use all the tools at your disposal.

3. Note if your change plan changes: why and how did you decide to deviate from the original plan?

4. Note if your initial change plan was incomplete: what was missing and how did you find out?

5. You have up to hour and a half to work on this.

6. When you have completed the change, notify the investigator.

[2]Hypertext link to `http://www.eclipse.org`
[3]Hypertext link to `http://www.eclipse.org/articles/index.html`
[4]Link to built-in Eclipse help system
[5]Hypertext link to `http://www.eclipse.org/search/search.cgi`